9 Bus Interface

A digital system uses *bus interfaces* to connect components with standardized communication protocols. For example, processor cores use one or more buses to communicate with memory and peripherals. Using a standard SoC bus makes it practical to attach off-the-shelf peripherals designed for that bus.

This chapter begins by exploring the principles driving SoC bus design. It then surveys a variety of standard SoC buses, including AHB-Lite used in Wally, as well as APB, TileLink, and others.

Chapter 20XREF describes off-chip peripheral interfaces using SPI and UARTs. High bandwidth off-chip interfaces such as DDR, PCIe, USB, and Ethernet are beyond the scope of this book.

The Wally processor from Chapter 4 used idealized instruction and data memories in the core. This chapter adds an external bus unit (EBU) that controls an AHB-Lite interface to an SRAM and ROM in the Uncore portion of the SoC, as highlighted in bright blue at the top of Figure 9.1.

<SIDEBAR>
The *Uncore* part of an SoC design is, as expected, everything except the processor core. It includes the bus interfaces, memory, and peripherals.
</END>

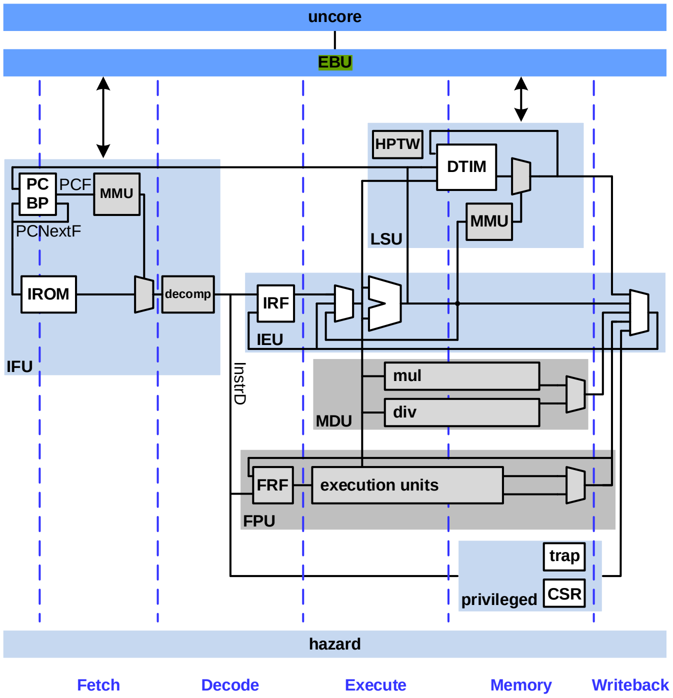


Figure 9.1 RV32I/RV64I high-level block diagram highlighting EBU and Uncore

9.1 Principles

A bus connects one or more *controllers* to one or more *peripherals* or *responders.* The peripheral might be a memory, such as SRAM, DRAM, or ROM, or an I/O device such as a timer, interrupt controller, general-purpose I/O (GPIO) device, serial port, data converter, etc. In most bus protocols, the controller sends an *address* to specify which peripheral and which memory location or register within that peripheral is being accessed. The bus also transmits *data* being read from or written to the peripheral.

<SIDEBAR>
Buses formerly used the nomenclature *master* and *slave*, and you will still see these terms in standards documents. These terms carry dehumanizing and derogatory connotations, so the industry is moving toward deprecating this language [Ellis21, OSHA22]. There is not yet agreement on new terminology. In 2021, ARM began using the terms *Manager* and *Subordinate* or *Requester* and *Completer*. In the absence of a new standard, this book uses the terms *controller* and *peripheral* (or *device*).
</END>

Figure 9.2 builds up a sequence of progressively more practical buses, showing the interface signals and timing. The simplest example of a bus is an *asynchronous read-only bus*, like the instruction memory interface from Figure 4.2XREF. The controller sends an *address* `A`, and a peripheral responds with *read data* `RD` within a bounded amount of time. Most memories and peripherals are *synchronous* for ease of implementation: they need the address to set up before a clock edge, and they return the read data after the clock edge. Peripherals other than ROMs and sensors need a read/write interface. They need to know what to write (*write data*, `WD`), where to write (`A`), whether to write (*write enable*, `WE`), and when to write (`clk`). Hence, a *synchronous read/write* bus sends the address (`A`), write data (`WD`), and write enable (`WE`) before the clock edge. When the write enable is asserted, the write takes place just after the clock edge. Read data (`RD`) is also returned after the clock edge.

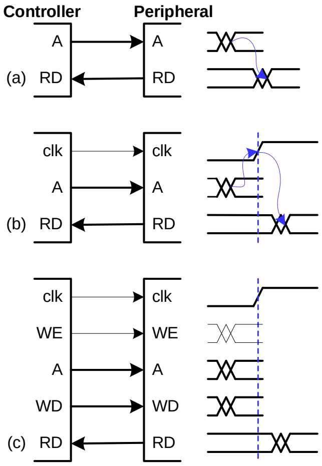


Figure 9.2 Basic bus interfaces: (a) asynchronous read-only, (b) synchronous read-only, (c) synchronous read/write

9.1.1 Bidirectional vs. Unidirectional Buses

The bus in Figure 9.2(c) has separate unidirectional read and write data ports. Pins on a chip and wires on a circuit board are relatively expensive, so designers were historically motivated to combine these two ports into a single bidirectional port, as shown in Figure 9.3. A tristate buffer, with alternating enable input polarity, is added on both the controller (left) and peripheral (right) sides. When `WE` is 1, the controller writes data to the peripheral; otherwise, the controller reads data from the peripheral.

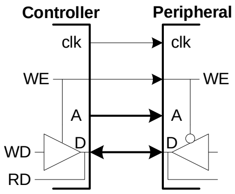


Figure 9.3 Synchronous read/write bus with bidirectional data port

In an SoC, far more wires are available, so their cost is lower. Moreover, long on-chip wires need repeaters to achieve reasonable speed, making bidirectional operation impractical. Hence SoC buses universally favor unidirectional buses, and tristate buffers are typically forbidden.

Unidirectional point-to-point off-chip buses also can run at a far higher speed than bidirectional multidrop buses. It is now cheaper to achieve a required amount of bandwidth by running a moderate number of wires at high speed in each of several unidirectional links than to run a larger number of wires at lower speed in a single bidirectional link. Section 9.1.6 discusses off-chip buses further.

9.1.2 Handshaking

A controller only needs to access the bus some of the time. To save power, it can provide a *Request* (REQ) line that is only asserted when it puts a valid address on the bus.

Some peripherals take more than one bus cycle to respond. For example, DRAM is fast for sequential reads but slow for reads to random addresses. A bus often provides an *Acknowledge* (ACK) or *Ready* (RDY) signal back from the peripheral indicating that the transfer is complete. This Request/Acknowledge sequence is known as *handshaking*. Figure 9.4 shows the Ready timing for a peripheral with a 2-cycle read access time. Observe how Ready (RDY) remains low for a cycle before rising on the second cycle when the read data (RD) is available.

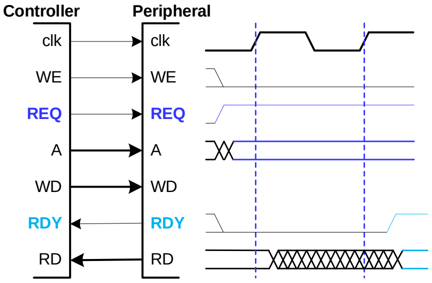


Figure 9.4 Handshake timing for multicycle access

9.1.3 Multiple Peripherals

In a practical system, the controller communicates with multiple peripherals. SoCs typically use *memory-mapped I/O* to select a particular peripheral based on the address range. The SoC contains an *address decoder* to produce select signals to choose which peripheral or memory to read, based on the address. The SoC *memory map* defines the addresses of each peripheral.

<SIDEBAR>
RISC-V generalizes the address decoder to a *Physical Memory Attributes* (PMA) unit that also determines whether the address is cacheable, what transfer widths are allowed, etc., as described in Section 8.2.2.1XREF.
</END>

Figure 9.5 shows buses that link multiple peripherals with *shared* and *point-to-point* data ports. Both versions require an address decoder. The shared bus allows multiple peripherals to drive the RD and RDY lines using tristates, as was shown in Figure 9.3. In contrast, the point-to-point design requires a read multiplexer to choose the read data (and ready signal) from the selected peripheral. Shared buses minimize the number of wires, but point-to-point buses are much faster and eliminate potential contention among peripherals. Shared buses have many problems in SoCs, including accurately modeling delay, verifying contention, and lacking repeaters to drive long wires. Therefore, SoC on-chip buses use point-to-point wiring.

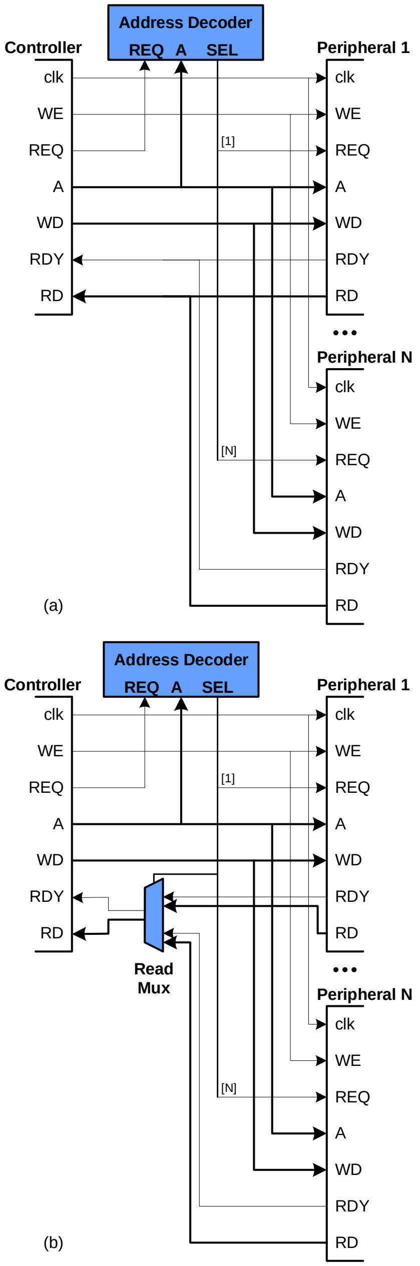

Address Decoder
REQ A SEL
Controller
clk
WE
REQ
A
WD
RDY
RD
Peripheral 1
clk
WE
REQ
A
WD
RDY
RD
[1]
•••
Peripheral N
clk
WE
REQ
A
WD
RDY
RD
[N]
(a)
Address Decoder
REQ A SEL
Controller
clk
WE
REQ
A
WD
RDY
RD
Peripheral 1
clk
WE
REQ
A
WD
RDY
RD
[1]
Read
Mux
•••
Peripheral N
clk
WE
REQ
A
WD
RDY
RD
[N]
(b)

Figure 9.5 Bus with multiple peripherals: (a) shared data ports, (b) point-to-point data ports and read multiplexer

9.1.4 Multiple Controllers

Larger systems also have multiple controllers that may share the same bus. One or more controllers may place a request on any given cycle by asserting their `REQ`. An *arbiter* picks among the requests by providing at most one Grant to one controller and chooses the address and write data from that controller, as shown in Figure 9.6. This synchronous bus uses a common clock for all components, which is not shown.

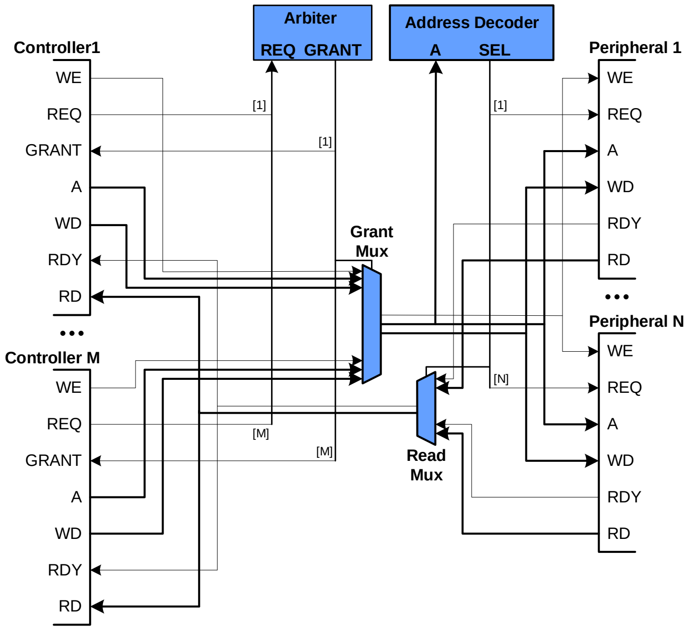


Figure 9.6 Bus with multiple controllers and arbitration

Arbiters can use various methods to pick which request to grant, including static, round-robin, time-division multiple access, and dynamic priority algorithms. In *static priority*, each controller has a fixed priority and the requesting controller with the highest priority is granted access. This could starve a low-priority controller if a high-priority controller is making continuous requests. In *round-robin*, access is granted in a circular manner so every controller will eventually get access. This is fair but could cause long delays on a critical request. In *time division multiple access* (*TDMA*), controllers are assigned time slots of varying length depending on their bandwidth requirements. In *dynamic priority*, the priority is adjusted in real time based on the system needs.

Systems with multiple controllers may need to *synchronize* between controllers so that only one can access a resource at a time. Synchronization is performed with *atomic* operations described in Chapter 13. These operations often read, modify, and then write a location in memory in such a way that no other controller can interrupt the operation. Thus, buses often have a *locking* signal so that a controller granted access can ensure it will complete the atomic operation before another controller gains access.

9.1.5 Other Bus Features

Additional bus features include support for multiple transfer modes (such as subword writes, burst transfers, and split transfers), cache coherency, and bridges between buses. High-performance systems may abandon buses altogether in favor of higher-speed networks. This section gives an overview of these features.

Buses often support various transfer modes. Most buses can write data smaller than the full word size. They provide either a Size or ByteEn signal that indicate which bytes to write. To reduce the overhead of repeated arbitration, buses also often define *burst* transfers consisting of a sequence of reads or writes, often to sequential addresses. Finally, when a data transfer is slow, it may tie up a bus for many processor cycles. For example, when a DRAM read is made to a new row, the bus may take many tens of cycles to respond. *Split transfer* buses tag the transfer with a unique ID and then open the bus for other operations. Some time later, the bus returns the data and tag to the controller that initiated the request.

Complex systems often use more than one type of bus interface. For example, a system may have a high-speed memory bus and a low-speed bus for legacy peripherals. A *bridge* interfaces two buses to translate the interaction from one bus to another and so that their protocols are each satisfied.

Some buses, such as ACE and TileLink Cached, incorporate *cache coherence* capabilities directly into the bus protocols to support *shared-memory multiprocessors* in maintaining *coherence* across their cache memories. Thus, any processor accessing a particular memory address will see the same data.

High-performance systems often need more bandwidth than a single bus can provide. For example, they may need to allow multiple processors to concurrently access multiple banks of level 3 cache memory. A *Network on Chip* (NoC) provides flexible and high-performance interconnect. Various NoC architectures include *crossbars*, where every component can communicate with any other component, *indirect networks*, in which components communicate through a series of switches, and *meshes* or *rings*, in which every component can communicate with its neighbors.

9.1.6 High-Speed Off-Chip Links

The bandwidth of off-chip links has grown along with processor performance. AT&T developed the first commercial modem in 1958 for the SAGE system that connected radar systems across the United States to detect intrusions by Soviet nuclear bombers. The modem ran over the phone lines at 110 baud (symbols per second), transmitting 80 bits per second (bps), which is 10 characters per second, between mainframe computers and teletype terminals.

Bandwidth has increased rapidly since those first links. In 2022, PCs are equipped with USB-C and Ethernet ports running at 10 Gbps to external devices and HDMI 2.1 ports supporting 48 Gbps using inexpensive cables. On the motherboard, a dual-channel DDR5-4800 memory system transmits approximately 500 Gbps between DRAM and a CPU, and PCIe 5.0 x16 transmits up to 1 Tbps between a graphics card and CPU. In data centers, Ethernet runs at 400 Gbps between cabinets. Wireless networks have also become very fast, with Wi-Fi 7 and 5G cellular theoretically capable of over 10 Gbps.

Although long-distance networks use fiber-optic cables driven by lasers, shorter links are primarily electrical over copper wires toggling as fast as possible. As the data moves along the copper at nearly the speed of light, transmission line effects are critical and the wires must be properly terminated with a matched impedance, making point-to-point links essential. The links use differential signaling, in which the data is represented as a difference in voltage between a pair of wires, rather than as a single-ended signal relative to a ground line. Differential signaling reduces crosstalk and electromagnetic interference because current is balanced, flowing out and back along the pair rather than returning over a shared ground line.

Some high-speed links such as DDR memory use source-synchronous clocking, in which a clock is sent along with a parallel bundle of data bits. The clock rises and falls each cycle, generally at a rate as fast as the transmission line can reliably support. If the memory bits only switched once per clock cycle, they would only use half of the available bandwidth of the line. Hence, double-data-rate (DDR) links pump data on both rising and falling edges of the clock to use the same toggle rate as the clock and fully saturate the transmission line’s capabilities.

Source-synchronous speeds are limited by mismatches in flight time between bits in the bundle. These mismatches are influenced by the line lengths, impedances, and noise. High-speed serial links avoid these mismatches by sending a serial bitstream over a differential pair encoded in such a way that the bitstream conveys both clock and data. The receiver performs clock/data recovery (CDR) to determine when to sample the serial stream, then decodes the data bits. For example, 8b/10b encoding sends 8 bits of data in a 10-bit symbol such that the transmitted bitstream toggles at least once every 5 bits even when the unencoded bitstream is a continuous pattern of 0s or 1s. The redundant bits can also provide error detection and correction. A phase-locked loop (PLL) in the CDR detects these transitions to align the sampling clock. Similarly, 10 Gb Ethernet uses 64b/66b encoding, PCIe 5 uses 128/130b encoding, and USB 3.1 uses 128b/132b to guarantee transitions while reducing the overhead. PCIe 5 x16 uses 16 serial lanes each operating at 32 GHz (64 GBps).

PCI Express, DDR memory, USB, and Ethernet each have specifications that are many hundreds of pages long, and they require a complex controller and physical interface (PHY) beyond the scope of this book. SoC designers purchase these controllers and PHYs as IP blocks. The controllers may be integrated onto an ASIC or FPGA and connected to the rest of the SoC via AXI or another standard on-chip bus. The SoC physical interface involves custom mixed-signal circuitry such as impedance-controlled differential output drivers and phase-locked loops. These are also sold by IP vendors.

All these sophisticated techniques for fast off-chip links cost chip area and power. Within the SoC, it remains more cost-effective to use wide buses to deliver high local bandwidth.

9.2 AMBA Buses

ARM's Advanced Microcontroller Bus Architecture (AMBA) is a leading family of open-standard on-chip buses used in both ASICs and FPGAs since 1996. The three major flavors are:

• **APB** (Advanced Peripheral Bus):	Simple unpipelined bus
• **AHB** (Advanced High-performance Bus):	Pipelined bus
• **AXI** (Advanced eXtensible Interface):	Multichannel, high speeds

Figure 9.7 shows an example system that uses AHB and APB. The high-speed CPU and on-chip memories use AHB, while the slower peripherals are on the simpler APB bus. AHB can also be used as an interface to off-chip memory.

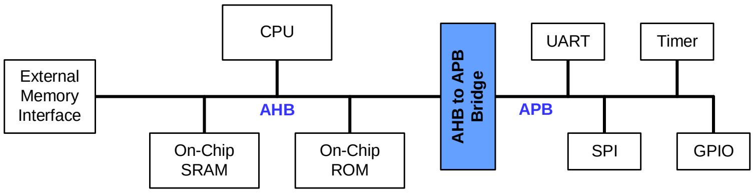


Figure 9.7 AHB/APB system

The following subsections describe APB [ABP21] and AHB [AHB21]. Both use two phases for address and data transfer, but AHB is faster because the phases of consecutive operations can overlap. AXI is now the dominant standard for systems with multiple controllers or high-speed DRAM interfaces, but the 500-page specification [AXI21] is beyond the scope of this chapter.

<SIDEBAR>
David Flynn (***-) was the original architect of ARM's synthesizable processors and the AMBA on-chip interconnect. He received the IEEE/RSE James Clerk Maxwell Medal in 2019 for these contributions. He also coauthored the *Low Power Methodology Manual* [Keating07], which defined industry-standard methods of clock and power gating. Flynn was an ARM Fellow and worked at the company from 1991-2018. He earned his BSc in Computer Science from Hatfield Polytechnic and his Doctorate in Electrical Engineering from Loughborough University.

Commented [h1]: Need permission pending David's review.

Ask for color to spice up bio
last contacted 6/8/22
ask for higher res picture if available

arm address no longer valid
pinged via LinkedIn 12/25/22
dflynn@ieee.org 1/14/23

</END>

9.2.1 APB

Figure 9.8 shows an APB bus connecting a controller (called the *Requester*) to one or more peripherals (called the *Completers*). The controller is usually an AHB-to-APB bridge. The standard APB signal names are similar to the generic ones from Figure 9.5, but are prefixed by P. The bus adds a PRESETn signal to reset all peripherals when asserted low. The address decoder, which generates the PSEL signals to the various peripherals, is part of the controller. The APB mux returns the data and ready signal from the selected peripheral. PADDR may be up to 32 bits, and PRDATA/PWDATA may be 8, 16, or 32 bits.

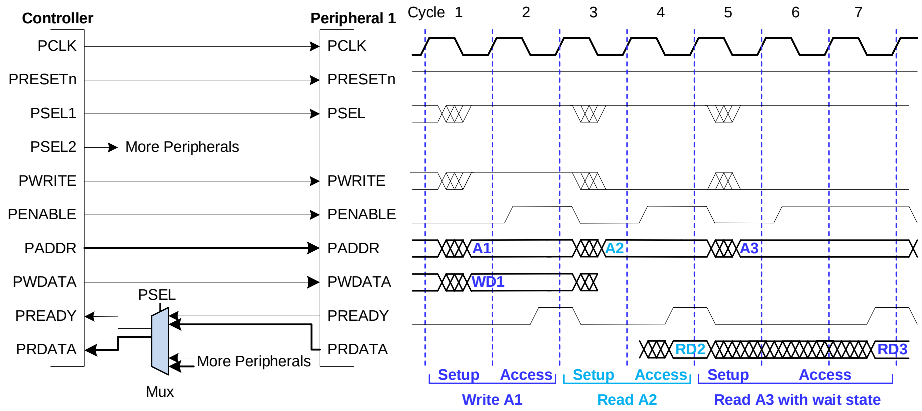


Figure 9.8 APB interface

Each transfer has two phases: *Setup* and *Access*. The Setup phase takes one clock cycle, during which the controller drives the PADDR, PWRITE, and PSEL signals on the bus along with PWDATA for a write. The Access phase takes one or more clock cycles, during which the controller holds these Setup phase signals steady and asserts PENABLE. The peripheral performs the read or write. Reads respond with PRDATA. The peripheral raises PREADY when it is done. Typically, this occurs in a single cycle, so the Access phase is one cycle long, but a slow peripheral could hold PREADY low for one or more *wait states* before responding. A peripheral that always responds in a single Access phase cycle can tie its PREADY high. If no peripheral is selected, the read mux asserts PREADY so the bus remains in a ready state.

The waveforms in Figure 6.6 illustrate read and write transfers. In cycles 1 and 2, the APB bus performs a write transfer with no wait states. In cycles 3 and 4, the bus performs a read with no wait states. In cycles 5-7, the bus performs a read with one wait state.

Not shown in Figure 9.8 are the optional PSTRB (APB store byte) and PSLVERR (APB peripheral error) signals. PSTRB[3:0] is a byte write enable. A peripheral may assert PSLVERR when an error occurs.

Example 9.1 Design a 32-bit General Purpose I/O (GPIO) peripheral that uses an APB interface. The peripheral has three 32-bit memory-mapped I/O registers: GPIOOUT, GPIOIN, and GPIOEN. GPIOOUT is the output register (for writing to the GPIO pins); GPIOIN is the read-only input register (for reading the values on the GPIO pins); GPIOEN configures the GPIO pins as outputs or inputs (1 = output, 0 = input), with each bit driving a tristate enable connected to the GPIO pins. These registers are mapped to a 2-bit address: GPIOOUT is at address 00, GPIOEN at 01, and GPIOIN at 10.

SOLUTION: Figure 9.9 shows the GPIO peripheral. The core of the peripheral is a tristate buffer that enables writing GPIOOUT onto the GPIO pins for each bit of GPIOEN[31:0] that is asserted. The pins can always be read onto GPIOIN. A 2:3 address decoder determines which register is being accessed. The GPIOOUT and GPIOEN registers are written at address 00 and 01, respectively, when PWRITE, PENABLE, and PSEL are all asserted. The GPIO multiplexer is controlled by the address to read the appropriate register. The peripheral has no wait states; thus, it always asserts PREADY.

<SIDEBARD>
ASIC and FPGA implementations use specially designed tristate buffers (also called drivers) capable of driving large capacitive PCB wire loads. They are instantiated in the top-level module, and therefore the gpio_apb module has three interface signals: GPIOEN, GPIOOUT, and GPIOIN.
</END>

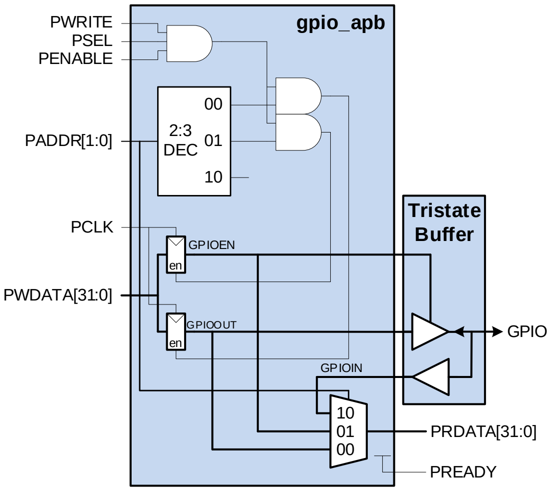


Figure 9.9 GPIO peripheral with APB interface

<SIDEBAR>
Figure 15.XREF shows a full-featured GPIO peripheral with an APB interface.
</END>

9.2.2 AHB

Like APB, AHB uses two phases of operation and takes at least two cycles for a transfer, with the option of additional wait states. However, AHB doubles the throughput by pipelining, that is, overlapping the two phases. AHB refers to the two phases as Adress (Adr) and Data phases, which are similar to APB's Setup and Access phases.

<SIDEBAR>
ARM uses inconsistent terminology between APB and AHB. The controller is called a *Requester* in APB and a *Manager* in AHB. The peripheral is called a *Completer* in APB and a *Subordinate* in AHB. Furthermore, the first phase is called *Setup* in APB and *Address* in AHB. The second phase is called *Access* in APB and *Data* in AHB.
</END>

<SIDEBAR>
AHB was originally developed to support multiple controllers and split transactions [AMBA99], with a simpler flavor called AHB-Lite omitting this support. Systems requiring these additional features soon migrated to the more capable AXI bus, leaving AHB-Lite as the favored flavor of AHB. ARM recognized this in the recent AHB5 specification, which is based on AHB-Lite with optional extra features related to security and exclusive transfers. This book will use the term AHB to refer to AHB5 and focuses on the required features that are common to AHB-Lite and AHB5.

</END>

Figure 9.10 shows an AHB bus connecting a controller (called the *Manager*) to one or more peripherals (called the *Subordinates*). The standard AHB signal names are similar to the ABP names from Figure 9.8, but are prefixed by `H`. The address decoder generates `HSELx` signals for the various peripherals and the AHB mux. `HADDR` may be up to 64 bits, and `HRDATA/HWDATA` are typically 32, 64, 128, or 256 bits, permitting wider transfers than APB. AHB adds an `HTRANS` signal to indicate the type of transfer, as summarized in Table 9.1. Typically, `HTRANS` is IDLE when the bus is not in use, or NONSEQ for ordinary transfers. Section 9.2.2.5 introduces SEQ burst transfers.

Table 9.1 HTRANS transfer types

HTRANS	Type	Description
00	IDLE	No data transfer
01	BUSY	Like IDLE, but in the middle of a burst
10	NONSEQ	Single transfer or first transfer of a burst
11	SEQ	Subsequent transfers of a burst

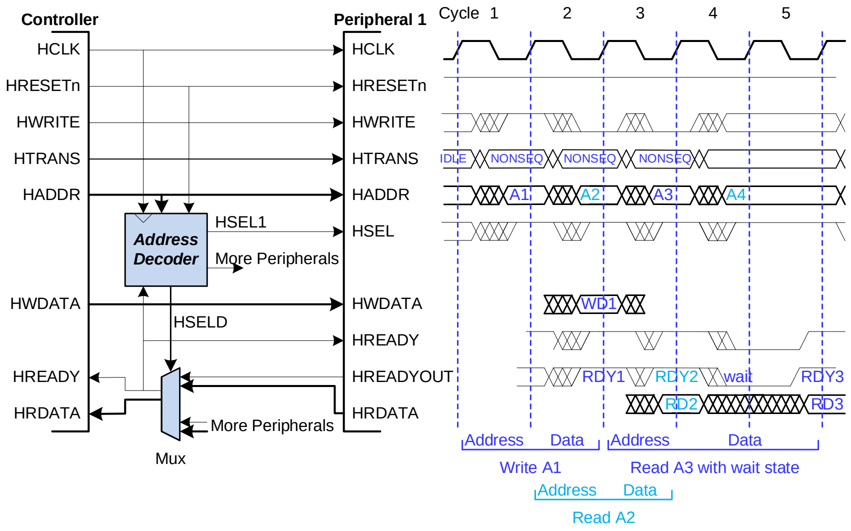


Figure 9.10 AHB interface in the Uncore

Figure 9.10 illustrates a write to address A1, then a read from A2, then another read from A3, which adds a wait state. In the address phase, the controller places the address `HADDR` and control signals `HWRITE` and `HTRANS` on the bus and the address decoder drives `HSEL` to the appropriate peripheral. In the data phase, the controller sends `HWDATA` or the peripheral responds with `HRDATA`. The peripheral also raises `HREADYOUT` to signal that the data phase is complete, or the peripheral may keep it low to insert one or more wait states before completing.

The AHB multiplexer chooses the `HREADYOUT` and possibly `HRDATA` from the selected peripheral. It chooses based on `HSELD`, a version of `HSEL` delayed to align with the Data phase. As in APB, when no `HSEL` signals are asserted, the AHB mux immediately asserts `HREADY` to prevent the bus from hanging. This is called the *default subordinate* behavior. AHB also sends `HREADY` to the peripherals to mark the completion of one transaction and start of the next. When a peripheral receives `HREADY` and `HSEL` and `HTRANS` is NONSEQ or SEQ, the peripheral must capture the Address phase inputs for the next transaction. Figure 9.11 shows a schematic of the address decoder and multiplexer, including the comparators to check `HADDR` against a peripheral's address range to find `HSEL`, the AND-OR muxing for `HRDATA` and `HREADY`, the HSELD delay register, and the default HREADY logic. The full Wally `pmachecker` in the memory management unit (Figure 8.21XREF) also considers access types and sizes in the address decoder.

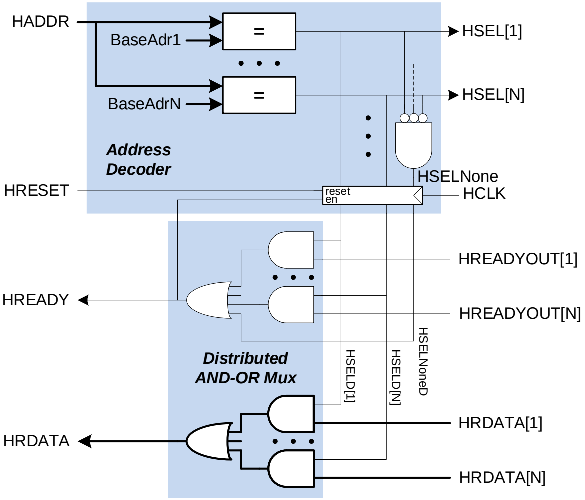


Figure 9.11 AHB address decoder and multiplexer

9.2.2.1 Additional AHB Signals

A full AHB link has several other signals shown in Figure 9.12 that were left out of Figure 9.10 for simplicity. The controller sends `HSIZE`, `HWSTRB`, `HPROT`, `HBURST`, `HMASTLOCK`, and optionally `HNONSEC`, `HEXCL`, and `HMASTER`. All of these are transmitted in the Address phase except `HWSTRB`, which is sent in the Data phase with `HWDATA`. The peripheral replies with `HRESP`, which is raised to indicate an error, and optionally `HEXOKAY` to indicate success of an exclusive transfer.

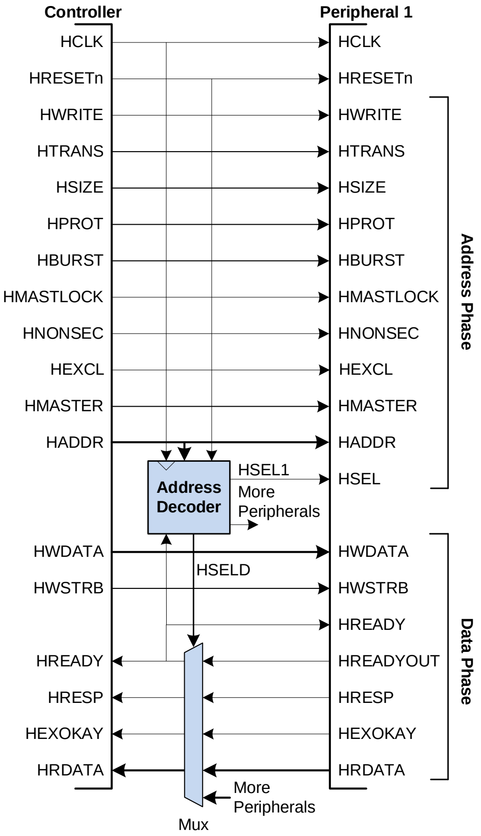


Figure 9.12 Full AHB Interface. Optional signals shown in blue

HSIZE indicates the size of the transfer, as defined in Table 9.2, not to exceed the width of the data buses. HWSTRB (write strobe) conveys byte write enables to write only certain bytes within a larger transfer. HPROT conveys optional information about protection, including cacheability, bufferability, and privilege/user and data/instruction modes. HBURST gives hints about burst transfers as will be discussed in Section 9.2.2.5. HMASTLOCK (master lock) and the optional HEXCL/HMASTER (exclusive/master) signals can be used for atomic operations. The optional HNONSEC signal identifies nonsecure transactions for systems with security extensions.

Table 9.2 HSIZE transfer sizes

HSIZE[2:0]	# Bits	Size
0	8	Byte
1	16	Half word
2	32	Word

3	64	Double word
4	128	
5	256	
6	512	
7	1024	

<SIDEBAR>
A little-endian memory system with a 32-bit bus could write a single byte, 0x42, to address 0x1002 in one of two ways. The first option sends a whole word and only enables one byte in the word to be written, and the second option makes a single-byte transfer.

Table 9.3 Two options for a 1-byte transfer on an AHB bus

Signal	Option 1	Option 2
`HADDR`	0x1000	0x1002
`HSIZE`	2 (4 bytes)	0 (1 byte)
`HWSTRB`	0b0100	0b0001
`HWDATA`	0x00420000	0x00000042

</END>

<SIDEBAR>
Many systems don't need all these other signals. The controller can tie `HSIZE` to the bus size (e.g., 2 for a 32-bit bus), `HPROT` = 3 (ignored by most peripherals), `HBURST` = 0 (no burst), and `HMASTLOCK` = 0, and omit optional signals. In a system where peripherals don't generate errors, the peripheral can tie `HRESP` to 0 and the controller can ignore `HRESP`.
</END>

9.2.2.2 AHB Peripherals

An AHB peripheral must perform several functions to interface with the bus:

- Detect new transactions
- Retime writes, so that `HWDATA` arrives in the Data phase rather than in the Address phase
- Read or write registers in the peripheral
- Produce HRDATA in the Address phase
- Generate the `HREADYOUT` acknowledgement

A peripheral starts a new transaction when the bus contains a valid transaction, the peripheral is selected, and the previous transaction is completing. The peripheral detects these conditions by asserting an internal signal, `initTrans`, when `HTRANS` is NONSEQ or SEQ, `HSEL` is asserted, and `HREADY` is asserted from the previous transaction. When `initTrans` asserts, the peripheral either reads or writes depending on the value of `HWRITE`.

<SIDEBAR>
`HTRANS[1]` indicates that the transaction is NONSEQ or SEQ.

</END>

When interacting with a peripheral that has no wait states, a controller reads from the peripheral on the rising edge after the Address phase and HRDATA is available during the Data phase. However, a write to the peripheral occurs on the rising edge after the Data phase because HWDATA is not available until the Data phase. Hence, at least HADDR and memWrite must be sampled at the end of the Address phase when HREADY is asserted to hold for writes in the data phase.

A peripheral may tie HREADYOUT to 1 if it never needs wait states, or it can delay asserting HREADYOUT until it is complete.

9.2.2.3 Example: AHB SRAM Interface

Figure 9.13 shows a single-port synchronous SRAM with an AHB interface. It asserts initTrans when a transaction is pending, the RAM is selected, and the previous transaction is completing. memRead or memWrite is asserted when initTrans is 1. A register delays HADDR and memWrite by a cycle to align with HWDATA arriving in the Data phase for writes and to retain the values for transactions with wait states. A multiplexer chooses the RAM address from HADDR on reads with no wait state, or from the delayed HADDRD for writes and for reads with one or more wait states.

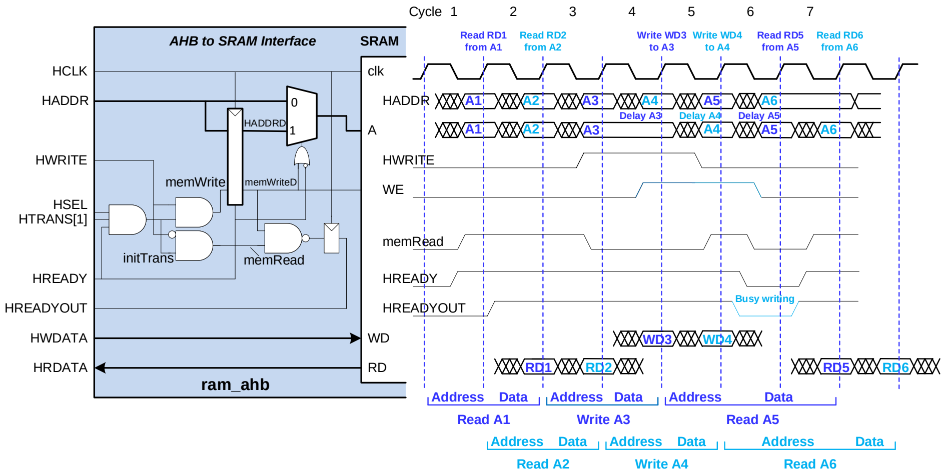


Figure 9.13 AHB SRAM interface

Reads occur on the rising edge at the end of the Address phase. Writes cannot occur until the rising edge at the end of the Data phase, so their address and write enable must be delayed a cycle. Thus, the SRAM can accommodate consecutive reads and consecutive writes, and a read followed by write, but not a write immediately followed by read because the SRAM has a single port and cannot accept the read and write addresses on the same cycle. Therefore, the SRAM inserts a single wait state by lowering HREADYOUT for one cycle for a write followed by read.

The waveforms in Figure 9.13 show two consecutive reads to A1 and A2, two consecutive writes to A3 and A4, and then two more reads to A5 and A6. The read after write has a wait state created by lowering HREADYOUT. As a result, HREADY and memRead must wait, delaying the read from A5.

<SIDEBAR>
A simple SRAM expects the write data at the same time as the address and control signals. The AHB protocol complicates the interface because write data arrives a cycle late. This timing is an artifact of AHB's roots as the native memory interface of early ARM processors [Flynn97]. For example, the ARM6 had a 3-stage pipeline in which the address arrived a cycle before the write data.
</END>

9.2.2.4 Example: AHB to APB Bridge

Figure 9.14 shows an AHB to APB bridge. The left side behaves as a peripheral on an AHB bus. The right side behaves as a controller on an APB bus. The bridge translates AHB transactions into APB transactions, adjusting the phases of the signals before returning HREADYOUT. The bridge supports a configurable number of peripherals. It expects select signals for each peripheral from the AHB address decoder. The bridge contains the APB read mux to choose the results from the selected peripheral. The bridge accepts the full width of the data buses and uses HWSTRB byte enables for subword writes.

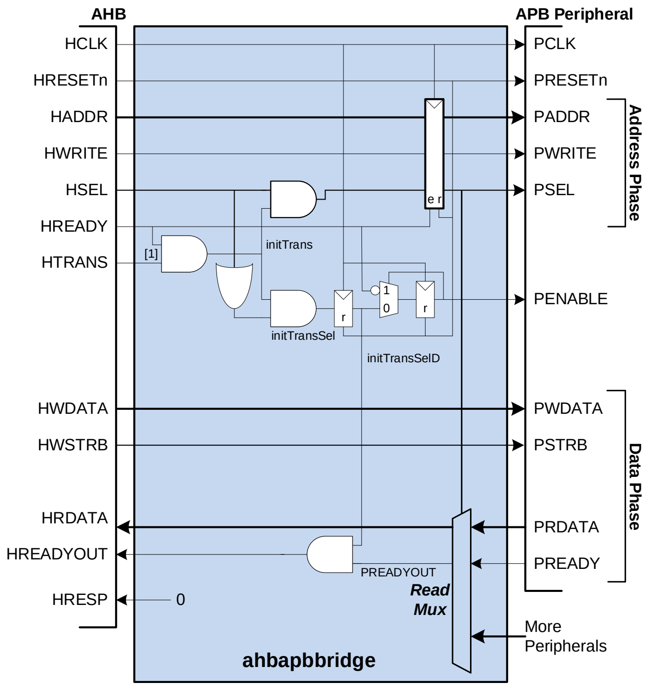


Figure 9.14 AHB to APB bridge

9.2.2.5 Burst Transfers

Some peripherals such as DRAMs are able to handle bursts of accesses to consecutive memory locations faster than accesses to random memory locations. AHB uses SEQ transactions and the HBURST signal to tell a peripheral that such a burst is occurring so that such peripherals can respond faster. The AHB standard defines various HBURST encodings for fixed and variable length, with addresses that increment or wrap around an aligned line. Simple peripherals ignore the SEQ and HBURST hints, and just respond to HADDR.

For example, Figure 9.15 shows a 4-beat write burst followed by a single read. The HBURST signal is set to INCR during the write burst to indicate addresses are incrementing. On the first beat, HTRANS is NONSEQ, and on subsequent beats, it is SEQ. HSIZE is 2 (010), so each beat transfers one 32-bit word, and the four beats transfer 128 bits all together.

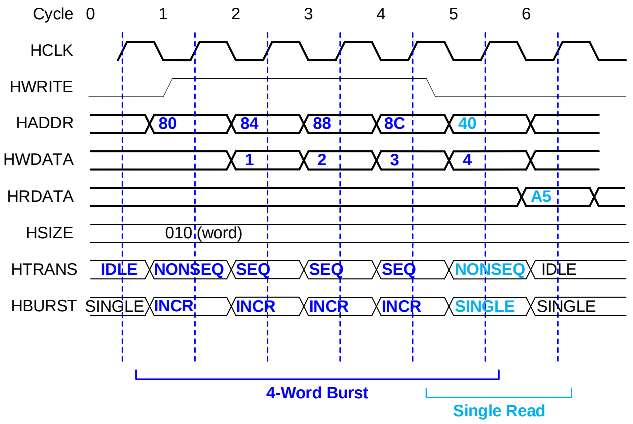


Figure 9.15 AHB burst transfer

9.2.2.6 Multilayer AHB

Multicontroller AHB systems now use independent AHB buses called layers for each controller. The layers are connected to peripherals through an interconnect matrix with decoders for each controller and arbiters managing access to each peripheral, as shown in Figure 9.16.

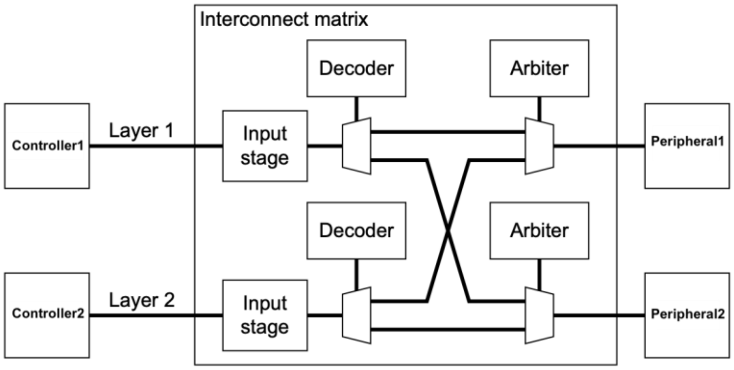


Figure 9.16 Multilayer AHB (adapted from [MultilayerAHB04])

9.3 Other SoC Buses

Several other buses are widely used on SoCs. SiFive developed the open-standard TileLink bus for RISC-V processors, and Section 9.3.1 surveys its major features.

Wishbone is an open-standard SoC bus developed by Silicore Corporation and maintained by OpenCores for portable intellectual property (IP) cores [Wishbone10]. It has similarities to AHB but doesn't delay write data by a cycle. It lacks the cache coherency features of TileLink. Wishbone is compatible with many free OpenCores of mixed quality, but it is not used as much in commercial systems.

The OpenHW group uses the Silicon Labs Open Bus Interface [OBI22] in several of its RISC-V processors. It resembles a simplified AXI interface, with an address and response channel.

CoreConnect, developed by IBM, is another commercial SoC bus. It includes a high-speed processor local bus (PLB) with a bridge to a simpler on-chip peripheral bus (OPB). It is available for free but requires a license from IBM. Xilinx FPGAs used CoreConnect IPs before switching to AXI; little CoreConnect IP is now available.

9.3.1 TileLink

TileLink is an open-standard SoC bus designed by SiFive for ease of implementation and to provide high speed and cache coherence [TileLink20]. It comes in three flavors:

- **TL-UL** (Uncached Lightweight): Only reads and writes
- **TL-UH** (Uncached Heavyweight): Adds bursts, atomic, prefetch
- **TL-C** (Cached): Adds cache block transfers

For example, Figure 9.17 shows the SiFive FU500 SoC with five processor cores connected over TL-C to the shared level-2 cache, and through a switch to TL-UL for each of the peripherals.

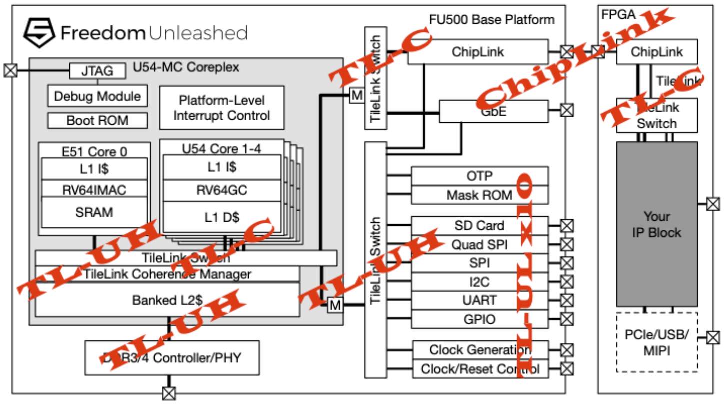


Figure 9.17 TileLink FU500 system example [Terpstra17]

Each component that sends or receives *messages* in a TileLink system is called an *agent*. TileLink *networks* use point-to-point links between controller and peripheral agents over unidirectional *channels*. In TL-UL, each link has two channels: a *request channel* from controller to peripheral, and a *response channel* from peripheral to controller. These channels are also called A (agent) and D (device), respectively. When a TileLink transfer is larger than the channel's data width, it is serialized into a burst of multiple *beats* on consecutive cycles. For example, a 32-byte transfer on an 8-byte bus takes four beats. The links form a directed acyclic graph, which prevents deadlock. Figure 9.18 shows an example of a TileLink network with two processors communicating with a cache and memory-mapped peripheral through a crossbar. TileLink-C adds three more channels for cache block transfers to avoid deadlocks while ensuring cache coherence.

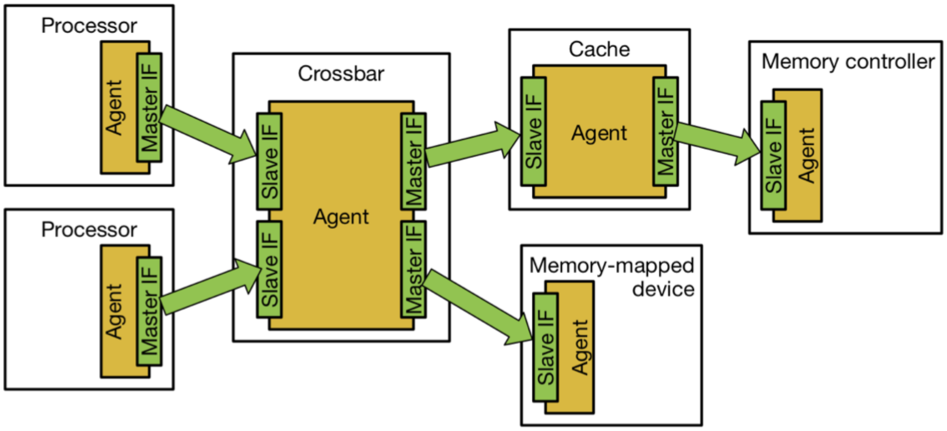


Figure 9.18 TileLink network example [TileLink20]

A key difference between AHB/APB and TileLink is that AHB/APB allow one controller to directly communicate with multiple peripherals, while TileLink allows only point-to-point links between a single controller and single peripheral. TileLink adds a switch component when it needs to connect multiple agents or peripherals. TileLink's point-to-point links facilitate high-speed SoC networks in which the link is pipelined to permit multiple messages in flight concurrently.

Error! Reference source not found. summarizes the signals in a TL-UL link. The synchronous clock and reset are shared between the request and response channels. The number of bits is system-dependent, with *a*-bit addresses and *w*-byte data. Similarly, the number of bits to specify the source IDs of the peripheral and controller are configurable. All signals go in the channel direction from sender to receiver (e.g., from controller to peripheral on a request channel) except `ready`, which goes the opposite direction to acknowledge a request.

Table 9.4 TileLink link signals

Request Channel	Response Channel	Bits	Meaning
`clock`		1	Shared synchronous clock
`reset`		1	Shared reset
`a_opcode`	`d_opcode`	3	Type of message on channel
`a_param`	`d_param`	2-3	Additional info for the opcode
`a_size`	`d_size`	z	*n* for 2^n-byte transfer
`a_source`	`d_source`	o	Controller source ID
	`d_sink`	i	Peripheral source ID

`a_address`		a	Target address
`a_mask`		w	Byte mask
	`d_denied`	1	Peripheral is unable to service request
`a_data`	`d_data`	8w	Write or read data
`a_corrupt`	`d_corrupt`	1	Corruption detected in data payload
`a_valid`	`d_valid`	1	Sender beat is available
`a_ready`	`d_ready`	1	Receiver can accept beat

Figure 9.19 shows the key request and response signals during TileLink transfers for a system with 8-byte data channels. Figure 9.19(a) shows two 32-byte read (Get) transactions on 8-byte channels. The controller places the Get opcode (4) on `a_opcode`, the size of 32 bytes (i.e., 5 for 2^5 bytes) on `a_size`, and asserts `a_valid`. The transfer takes an indefinite amount of time, but the peripheral eventually replies with the AccessAckData opcode (1) for four beats to transfer the 32 bytes (8 bytes per beat) while it asserts `d_valid`. The controller then makes another Get request, which is extended for a cycle in this example until `a_ready` is asserted by the peripheral. This time, the peripheral responds more quickly with another AccessAckData and four more beats of data.

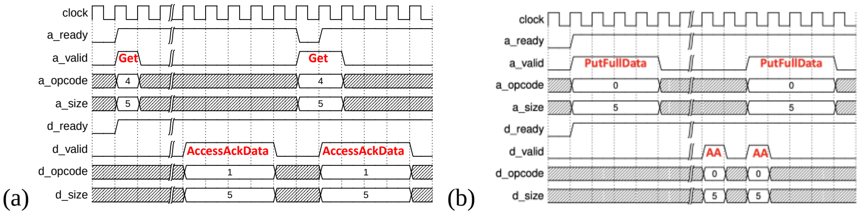


Figure 9.19 TileLink Transfers: (a) read, (b) write [Terpstra17]

Similarly, Figure 9.19(b) shows two 32-byte write (Put) transactions. The controller places the PutFullData opcode (0) on `a_opcode`, the size of 32 bytes on `a_size`, and asserts `a_valid` for four beats. The transfer takes an indefinite amount of time, but the peripheral eventually replies with the AccessAck opcode (0) for a single beat while asserting `d_valid` to acknowledge the completion of the entire (32-byte) write. The controller then makes another PutFullData request. This time, the peripheral responds immediately with another AccessAck for one beat.

9.4 Test Plan

The bus has primarily been tested through normal use of memory and peripherals.

The `ahb_ram_latency_{0/1/2}_burst_en_{0/1}` tests stress the variable latency AHB interface and burst mode operation. They run the `fadd_b11-01` test (see Section 13.XREF), where the active data exceeds the cache sizes and, thus, generates memory traffic. The testbench configures the RAM latency to 0, 1, or 2 cycles and enables or disables burst mode cache line fills (see Section 9.XREF).

A more comprehensive test plan would be beneficial.

9.5 Wally Implementation

The basic pipelined processor core from Figure 4.11XREF has no bus interface. This section adds an *Uncore* module with memory and peripherals communicating over AHB. The memory and peripherals are collectively called AHB *devices*. When the `BUS_SUPPORTED` configuration variable in `wally-config.vh` is 1, Wally adds an AHB interface to the LSU and IFU to access memory or peripherals beyond the DTIM or IROM. It also adds an *external bus unit* (EBU) that arbitrates between requests from the LSU and IFU and passes the selected request onto the Uncore AHB.

Wally's AHB interface is shown in Figure 9.20. The AHB, EBU, and Uncore modules are added when BUS_SUPPORTED = 1. The `wallypipelinedsoc` top-level module comprises the `wallypipelinedcore` and `uncore` modules. `wallypipelinedcore` includes the processor core, which includes the LSU, IFU, and EBU (modules `lsu`, `ifu`, and `ebu`). The Uncore module (`uncore`) includes on-chip RAM (`ram_ahb`) and ROM (`rom_ahb`), and an optional AHB interface to off-chip memory. Table 9.5 summarizes the configuration parameters for these bus-based memories. The Uncore also has an AHB-to-ABP bridge (`ahbapbbridge`) to various peripherals described in Chapter 15. For performance or for the rv{32/64}i configurations with no bus, the LSU and IFU optionally contain tightly integrated DTIM and IROM. Chapters 9 and 10 will add optional caches instead.

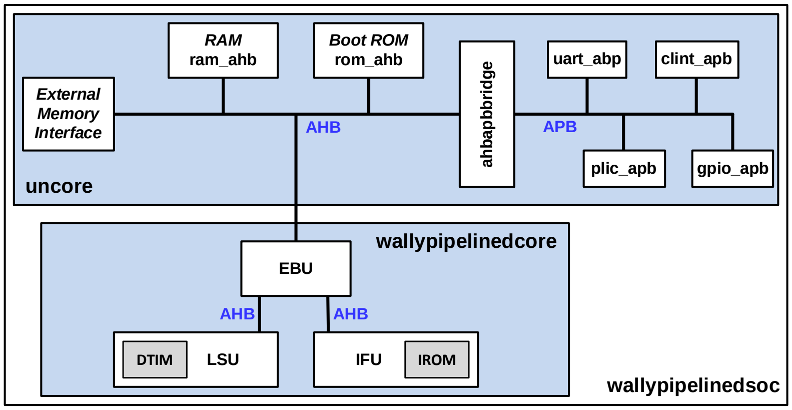


Figure 9.20 Wally AHB interface and top-level system diagram

Table 9.5 Wally bus memory configuration options

Configuration Parameter	Meaning	Typical Address
UNCORE_RAM_SUPPORTED	RAM in the Uncore on AHB	80000000

BOOTROM_SUPPORTED	ROM in the Uncore on AHB for bootloader	1000
EXT_MEM_SUPPORTED	AHB interface to RAM outside the Uncore (e.g. via a DDR memory controller on an FPGA)	80000000

To enable bus access, the LSU and IFU add AHB interfaces that include an FSM to sequence the AHB phases and a register to capture HRDATA. The core adds an external bus unit (EBU) to arbitrate between LSU and IFU requests. Wally adds a new top-level `wallypipelinedsoc` module containing the `wallypiplinedcore` and the Uncore memories and peripherals. Wally optionally supports the synchronous IROM and DTIM memories for fast code and data storage because the bus access to external memory is slower than tightly integrated memory.

9.5.1 LSU/IFU AHB Interface

The previous LSU and IFU described in Chapter 4 included DTIM and IROM memory in the core. Figure 5.14XREF added optional biendian support. This section describes a configuration that uses a bus interface instead, with access to Uncore AHB devices via the EBU. The bus `HRDATA` and `HWDATA` widths are XLEN.

<SIDEBAR>
The rv32imc*** configuration has a DTIM and IROM as well as a bus to Uncore peripherals.
</END>

Figure 9.21 shows the LSU with a bus interface. As compared to Figure 5.14XREF where the LSU had a DTIM, this updated LSU keeps the subword and biendian support, but removes the DTIM and adds the AHB signal pins and an `ahbinterface` module with fetch and write buffers and a `busfsm` control state machine. The fetch buffer holds the HRDATA read from the bus, and the write buffer holds the values driving HWDATA and HWSTRB onto the bus. The `busfsm` walks through the phases of the AHB transaction and generates AHB control signals and the fetch buffer enable at appropriate times. There is no need for the DTIMAdr multiplexer of Figure 4.13XREF because the bus transaction starts in the Memory stage for both loads and stores.

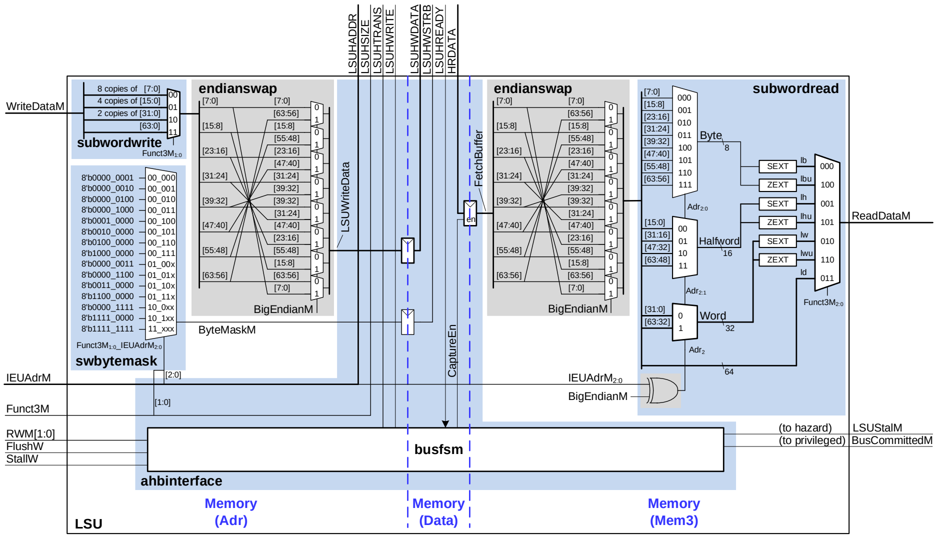


Figure 9.21 LSU with `ahbinterface` (`lsu` module configured with ``BUS_SUPPORTED` = 1, ``DTIM_SUPORTED` = 0, ``DCACHE_SUPPORTED` = 0)

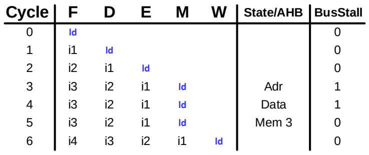
<SIDEBAR>
Several AHB control signals are omitted from Figure 9.21 for brevity. `HBURST` is hardwired to 0 for now; Section 9.XREF will introduce burst-mode cache line fetches and writebacks. `HPROT` is tied to 4'b0011 and `HMASTLOCK` is tied to 0 because protection and bus locking are not supported. `HRESP` is ignored because the memory system does not produce any error conditions. The Uncore module generates `HSEL` based on `HADDR` and configured memory ranges for memories and peripherals, as will be discussed in Section 9.5.3.1.
</END>

The IFU similarly replaces the IROM with an `ahbinterface` that has a fetch buffer and bus read state machine. It does not support subword accesses and endian swapping. `HSIZE` is always 010 to fetch 4-byte instructions.

Recall from Table 4.7XREF that an LSU DTIM access involves work in the Execute, Memory, and Writeback stages, but only stalls for hazards. In contrast, an access to bus-based memory always takes at least two extra cycles, as shown in Figure 9.22 for `ld` followed by additional instructions (i1, i2, …). The LSU stalls the CPU during these extra cycles by setting `BusStall` = 1, so the LSU and AHB behave as a multicycle (nonpipelined) system. Table 9.6 summarizes the steps the LSU takes to access the AHB bus. The Memory stage is stalled so that it takes at least three cycles. During the first cycle, the AHB Adr phase, `HREADY` may be low indicating the BUS cannot take a request. The LSU holds the AHB Adr phase signals steady until `HREADY` is high. The second cycle is the AHB Data phase, where the device reads or writes. A slow device may stretch phase this by delaying `HREADY`. The third cycle buffers `HRDATA` and muxes the bits for either subword accesses or if needed to reorder bytes for endianness.

<SIDEBAR>
The AHB bus is inherently pipelined with the address phase of the current request overlapping with the data phase of the previous request. Wally could leverage the pipelining to reduce data and instruction memory access latency. However, it would require modification to the standard 5-stage pipeline. For example, the LSU issues the memory request at the start of the Memory stage and expects ReadDataM by the end of the cycle. To leverage AHB's pipelining the LSU would need to issue the address a cycle earlier in the Execute stage. This increases the critical path along IEUAdrE. Chapter 8 adds an MMU to the Memory stage to translate virtual addresses and check for address exceptions. This logic would need to move to Execute stage to accommodate the earlier AHB request, which has major critical path implications. Alternatively the AHB request could begin in the Memory stage and complete in the Writeback stage adding an extra cycle to the Load Delay Hazard. Ultimately these variations could be implemented; however, Amdahl's law tells us to make the common case fast. Most bus accesses will be filtered by the instruction and data caches, so any pipelining advantages are limited by infrequent bus access.

The Data and Memory 3 phases could be merged, but this might increase the cycle time of the processor. In a processor with a DTIM or cache, bus accesses are infrequent, so it is better to have a fast cycle time than a lower bus latency.
</END>



Cycle	F	D	E	M	W	State/AHB	BusStall
0	ld						0
1	i1	ld					0
2	i2	i1	ld				0
3	i3	i2	i1	ld		Adr	1
4	i3	i2	i1	ld		Data	1
5	i3	i2	i1	ld		Mem 3	0
6	i4	i3	i2	i1	ld		0

Figure 9.22 `ld` instruction using AHB bus

Table 9.6 LSU AHB access steps

Stage	**Read**	**Write**
Execute	Compute memory address `IEUAdrE` = base address + offset	
Memory1 / AHB Adr Phase	Drive `HADDR` = `IEUAdrM`, other AHB control signals Writes also perform subword write muxing	
Memory2 / AHB Data Phase	Read `HRDATA` synchronously from AHB device	Drive `HWDATA` onto AHB
Memory 3	Capture `HRDATA` into `ReadDataM` at start of cycle, then perform subword read and endian muxing	Write `HWDATA` into device
Writeback	Write `ReadDataW` into register file	nothing

The LSU AHB bus interface (busfsm) is a three-state FSM controller shown in Figure 9.23 that sequences the steps from Table 9.6 for both stores and loads. State names match the AHB specification's Adr and Data phases with the added third state called Memory3 (Mem3). All memory requests start in the Adr Phase. If read or write is requested, busfsm asserts BusStall to stall the pipeline while the bus services the request. Once the bus is ready, busfsm asserts HTRANS = 2 (NON_SEQ) to initiate a single AHB read or write, and advances to the Data phase. While in the Data phase, the FSM asserts BusCommitted to delay the Privileged Unit taking interrupts until the bus transaction completes. Once the bus raises HREADY to complete the Data phase, busfsm advances to Memory 3 and raises CaptureEn on reads to enable the fetch buffer to sample HRDATA. While in Memory 3, BusStall is released so the pipeline can advance. The StallW, BusStall, and BusCommitted signals are discussed further in Section 9.5.5.

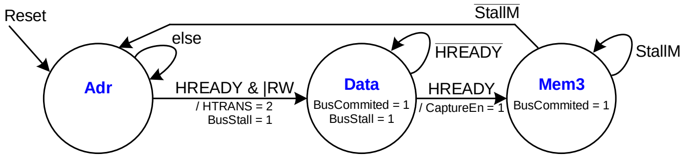


Figure 9.23 busfsm state machine. Stall = StallW for the LSU and StallF for the IFU.

<SIDEBAR>
Section 9.2.1XREF adds burst-mode to the bus interface for fast cache line fills.
</END>

<SIDEBAR>
The processor runs slowly without some sort of local memory because each instruction or data access takes at least 3 cycles over the bus. Chapter 9 describes the full LSU with a cache and bus. The cache services most accesses, while the bus fetches cache misses from RAM and accesses peripherals in the Uncore.
</END>

The IFU interface is similar but is 32-bit little-endian read-only. Thus, it drops HWDATA, HWRITE, HWSTRB, and the subword and endian logic, as shown in Figure 9.24. The IFU busfsm receives StallD instead of StallW.

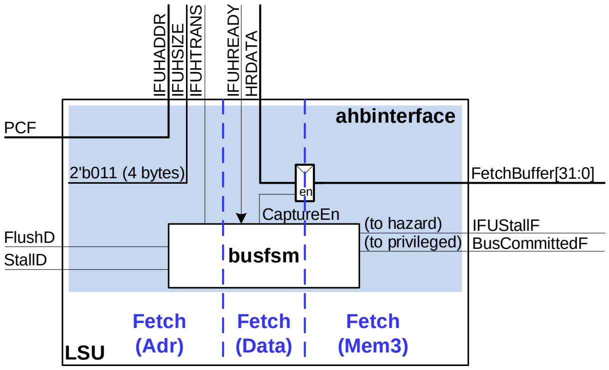


Figure 9.24 IFU `ahbinterface` (`ifu` module configured with `` `BUS_SUPPORTED `` = 1, `` `IROM_SUPORTED `` = 0, `` `ICACHE_SUPPORTED `` = 0)

9.5.2 External Bus Unit

The external bus unit (EBU) implements a subset of the multilayer AHB protocol to arbitrate between the IFU and LSU AHB controllers, giving the LSU priority. The selected controller's request is placed on the bus to the Uncore. As shown in Figure 9.25, the EBU consists of an arbitration FSM, muxes to choose the selected controller, and registers to retain the other controller's request. Section 9.1.1.3 will extend the EBU to handle burst transactions.

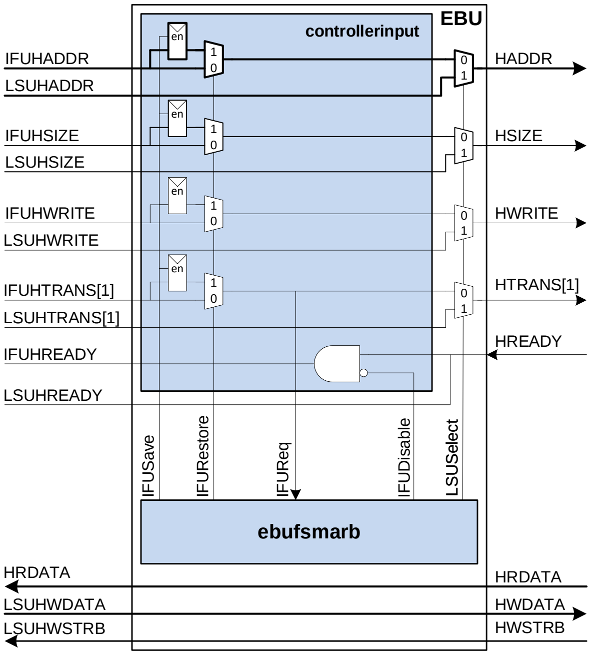


Figure 9.25 External Bus Unit (EBU)

HRDATA and HWDATA bus widths are AHBW, which is set to the natural word size XLEN in Wally configurations. Section 8.2.1XREF discusses virtual and physical addressing and explains that RISC-V uses 34-bit physical addresses for RV32 and 56-bit physical addresses for RV64. HADDR is a physical address of width PA_BITS (34 or 56). Recall that programs use 32 or 64 bit addresses. In systems with no memory management unit, RV32 appends two leading 0s, and RV64 drops the 8 most significant bits. Systems with a memory management unit supporting virtual memory translate XLEN-bit virtual addresses to PA_BITS physical addresses.

The `ebufsmarb`, shown in Figure 9.26, arbitrates when both the LSU and IFU make a request at the same time. It always gives priority to the LSU to prevent deadlocking the pipeline. Only two states are required, an IDLE state to indicate either 1 or 0 requests are active and an ARBITRATE state which selects the LSU. It is a Mealy machine with outputs that are a function of the current state and inputs. When two requests occur in the IDLE state, the FSM transitions to ARBITRATE and selects the LSU. Because the IFU detects `HREADY` = 1, it assumes the Adr phase of the request was taken and moves on to the Data phase. The `ControllerInput` module must save the IFU's Adr phase `HADDR` and control signals for later. Data phase signals are inherently delayed, so they need not be stored and replayed. Once the device completes the memory request, `HREADY` asserts and the FSM returns to IDLE. Upon returning to IDLE, the IFU Adr phase is replayed by driving the stored request onto the bus.

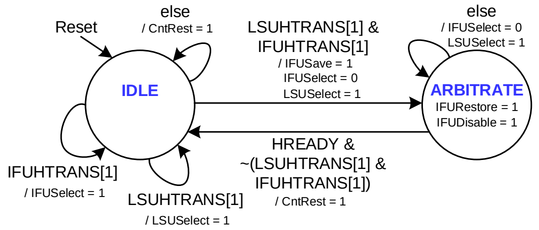


Figure 9.26: EBU FSM arbiter (ebufsmarb), with the LSU having priority over the IFU

Figure 9.27 shows the timing of a concurrent IFU and LSU read followed by two more IFU reads. In cycle 0, both the IFU and LSU make requests by driving IFUHTRANS and LSUHTRANS to NON_SEQ (2). The FSM chooses the LSU, raising SelLSU (select LSU) to grant the LSU bus access and drive the LSU Adr phase signals onto the bus. For example, HADDR = LA1. The IFU request is saved inside the controllerinput register with IFUSave = 1. The FSM transitions to Arbitrate in cycle 1. HREADY is 1, indicating that the device has returned HRDATA. LSUHREADY asserts to capture HRDATA into the LSU. Meanwhile, the EBU selects the saved IFU request's Adr Phase signals (IA1) to drive the bus. IFUHREADY is 0 to continue stalling the IFU. In cycle 2, the device responds to the IFU request with HRDATA for IA1 and HREADY = 1. The FSM returns to IDLE now that the LSU is done. Two more requests originate from the IFU, so no arbitration is needed for cycles 3 and 4.

<SIDEBAR>
When the data cache is added to the LSU, cache misses will generate AHB burst requests. Burst operations require the EBU to count each word of the transaction and ensure the lower priority IFU is not interrupted during a burst.
</END>

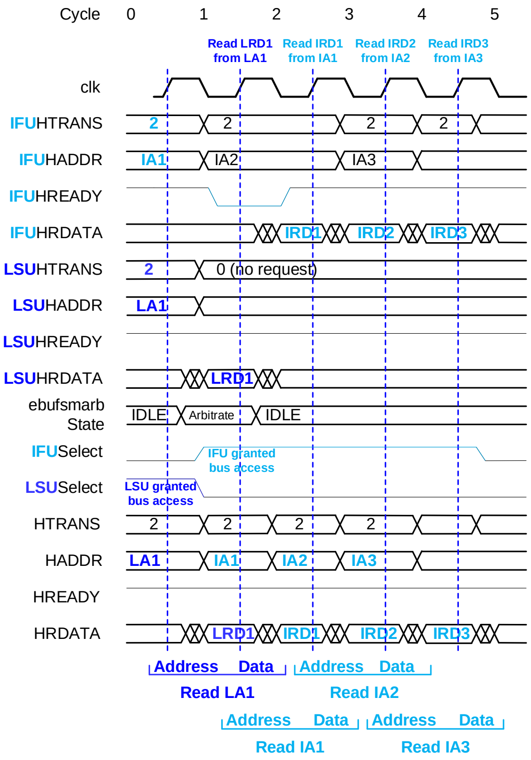


Figure 9.27 Timing diagram for IFU and LSU arbitration of concurrent requests

9.5.3 System-on-Chip

As was shown in Figure 9.20, the `wallypiplinedsoc` (System on Chip) module includes both the processor core and the Uncore module that contains memory, peripherals, and an external memory interface connected to the core via the bus. Chapter 15 describes the peripherals further. The external memory interface could be an off-chip AHB device such as a DDR4 memory controller. The Uncore module also contains address decoding and device multiplexing. Figure 1.30XREF showed Wally's default memory map.

9.5.3.1 Address Decoding and Device Mux

As was shown in Figure 9.12, AHB requires an address decoder and multiplexer to select and choose from among multiple devices. Figure 9.28 shows the Uncore, which includes the address decoder, mux, and several peripherals, emphasizing the connections of `HADDR`, `HREADY`, `HREADYOUT`, and `HSEL`.

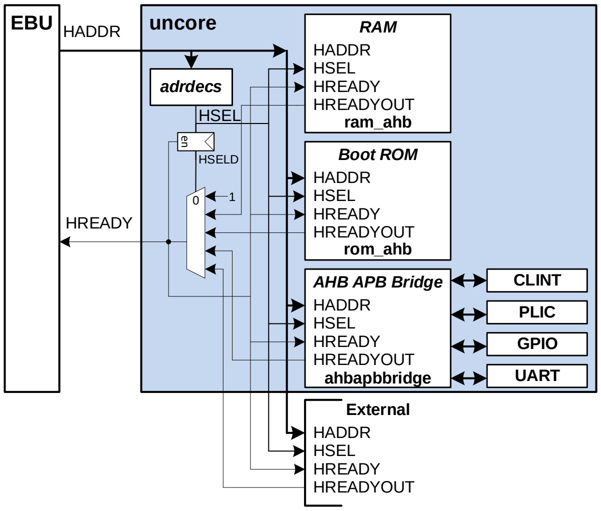


Figure 9.28: Uncore interface from EBU to devices

Recall that AHB is pipelined into Adr and Data phases, and that each phase ends with an HREADY signal being asserted by the selected device. Because of pipelining, the HREADY signal in the Adr phase of one transaction comes from the Data phase of the previous transaction.

The decoder uses the configuration parameters <DEVICE>_SUPPORTED, <DEVICE>_BASE, and <DEVICE>_RANGE to determine which devices (such as BOOTROM, UNCORERAM, GPIO, etc.) are supported and where they appear in the memory map.

The adrdecs module receives HADDR in the Adr phase and produces a one-hot HSEL signal, with one bit per device. The module contains an address comparator for each device; an implementation is shown in Section 8.4.1.1XREF. HSEL[0] is asserted when HADDR doesn't match any device address. When the previous transaction's Data phase completes by the peripheral asserting HREADY, the current transaction enters the Data phase and latches HSEL to HSELD. The currently selected device reads or writes during the Data phase and raises its HREADYOUT when complete. The HREADY multiplexer picks HREADYOUT from the selected device (based on HSELD) and drives HREADY back to the EBU and also to all the devices. If no device is selected, the bus is idle and immediately returns HREADY to avoid locking up. The result multiplexer, not shown in this diagram, also selects HRDATA and HRESP from the selected device.

<SIDEBAR>
Upon reset, both HSEL[0] and HSELD[0] must be set to 1 to assert HREADY and, thus, prevent the bus from locking up.
<END>

Example 9.2 Suppose an AHB bus is initially idle, then reads from the ROM immediately followed by a read from the RAM, and then returns to idle. Suppose the ROM and RAM are slow, with 2 and 1 extra wait states, respectively. Let the RAM and ROM be devices 1 and 2. Sketch a timing diagram of `HSEL`, `HSELD`, `HREADYOUT` (from the ROM and RAM), and `HREADY`.

Solution: Figure 9.29 shows a timing diagram. During the IDLE state, `HSEL[0]` is asserted to indicate no request is present. When the ROM transaction Adr phase begins in cycle 1, the controller asserts `HSEL[1]` to indicate a ROM access. `HREADY` was 1 because the bus was idle. The ROM Data phase begins in cycle 2, but the slow ROM does not assert `HREADYOUT` until cycle 4. In cycles 2-4, the mux selects `HREADY` as the ROM's `HREADYOUT`. Meanwhile the RAM Adr phase begins in cycle 2 and asserts `HSEL[2]` to indicate a RAM access, but the access cannot begin until cycle 4 when `HREADY` rises. In cycle 5, the RAM Data phase begins, but the slow RAM does not assert `HREADYOUT` until cycle 6. In cycles 5-6, the mux selects `HREADY` as the RAM's `HREADYOUT`. Meanwhile, the bus returns to idle and asserts `HSEL[0]` to indicate no requests in cycles 5 and beyond. `HSELD` indicates the selected peripheral in the Data Phase.

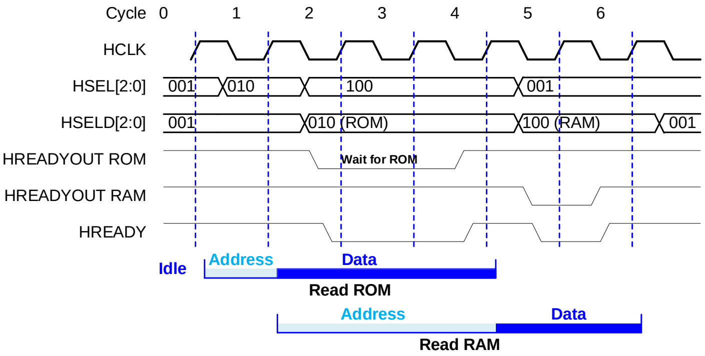


Figure 9.29: AHB timing diagram of consecutive reads to different devices with wait states

9.5.4 Combined Bus and Tightly Integrated Memories

An embedded processor typically doesn't have caches but instead has a small IROM and DTIM in the core as well as a bus to peripherals or external memories. The address decoder must distinguish between accesses to these tightly coupled memories versus bus accesses, and use a mux to read accordingly.

Wally handles this decoding in a memory management unit (MMU) using a physical memory attribute (PMA) checker, as described in Sections 8.2.2.1XREF and 9.3.1XREF. Figure 9.30 shows a diagram of the LSU with both DTIM and AHB bus. The PMA checker's address decoder outputs `SelDTIM` which controls a multiplexer to select between the DTIM's read data

(`DTIMReadDataM`) or the AHB's fetchbuffer (`FetchBuffer`). `SelDTIM` also demuxes `RWM` to either the DTIM or `busfsm`.

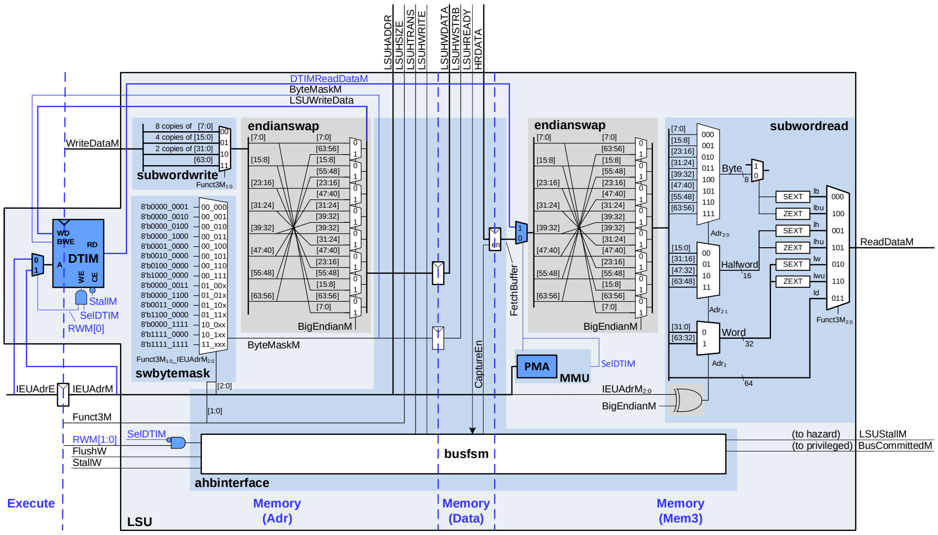


Figure 9.30: LSU with DTIM and bus

Figure 9.31 abstracts the LSU to emphasize the data and address paths. The `busfsm` is hidden in the `abhinterface`. The processor stalls in the Memory stage during the multicycle bus access. The three (or more) AHB cycles (Adr, Data, Mem3) are not shown, and the core resumes in the Memory stage when the stall is released during Mem3.

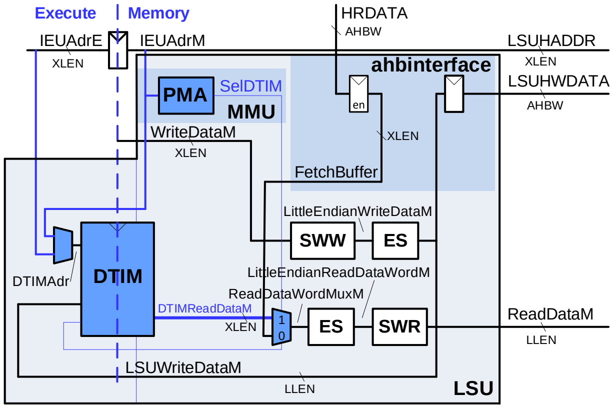


Figure 9.31 Abstracted LSU with DTIM and bus

The instruction fetch unit (IFU) uses the same approach to select between IROM and the Bus. A separate PMA checker outputs `SelIROM` to select the IROM or the Bus, Figure 9.32. Unlike the LSU's DTIM, the IROM only reads instructions. `PCNextF` directly drives the IROM read address port.

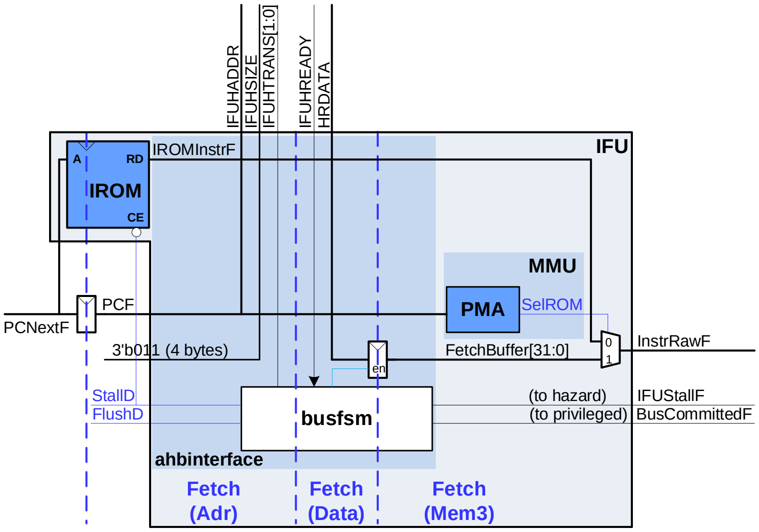


Figure 9.32 IFU with IROM and bus

9.5.5 Bus Hazards

Each AHB interface in the IFU and LSU takes multiple cycles to complete a bus transaction. The `busfsm` from Figure 9.23 must *interlock* with the Hazard Unit to correctly stall the pipeline and delay interrupts occurring in the middle of a bus transaction.

<SIDEBAR>
An *interlock* disallows certain states to occur that would conflict. For example, an interrupt that occurs during a bus transaction must be delayed until after the bus transaction completes.
</END>

The LSU or IFU `busfsm` generates `BusStall` while the bus is in the midst of a transaction. `BusStall` is asserted when the bus transitions from the Adr phase into the Data phase and remains high until the Data phase is completed. When `BusStall` is lowered the Hazard Unit un-stalls the pipeline and the IEU captures the load result from `ReadDataM`.

The LSU raises `LSUStallM` when its `BusStall` is asserted. Similarly, the IFU raises `IFUStallF` when its `BusStall` is asserted. The Hazard Unit raises `StallW` to stall the entire pipeline when either bus stalls, as shown in Figure 9.33. Traps take priority and should flush the pipeline rather than processing the LSU/IFU stall. However, these traps only occur at the start of the transaction. The `CommittedM/F` signals, discussed later in this section, postpone interrupts arriving in the midst of a transaction.

Recall that a stall at a given stage will result in a stall of all previous stages. Thus, a stall in the Writeback stage stalls the entire pipeline stalls.

<SIDEBAR>
The bus stall could be expressed as:

```
StallWCause = (IFUStallF & ~FlushFCause | LSUStallM) & ~FlushWCause
```

because a Fetch flush has priority over a IFU stall (`IFUStallF`) and a Writeback flush has priority over a LSU stall. `FlushFCause` is not explicitly defined but is equivalent to `FlushDCause` because a flush clears all previous pipeline stages. `FlushWCause` is a strict subset of `FlushDCause`. Hence the bus stall simplifies to:

```
StallWCause = (IFUStallF & ~FlushDCause) | (LSUStallM & ~FlushWCause)
```
</END>

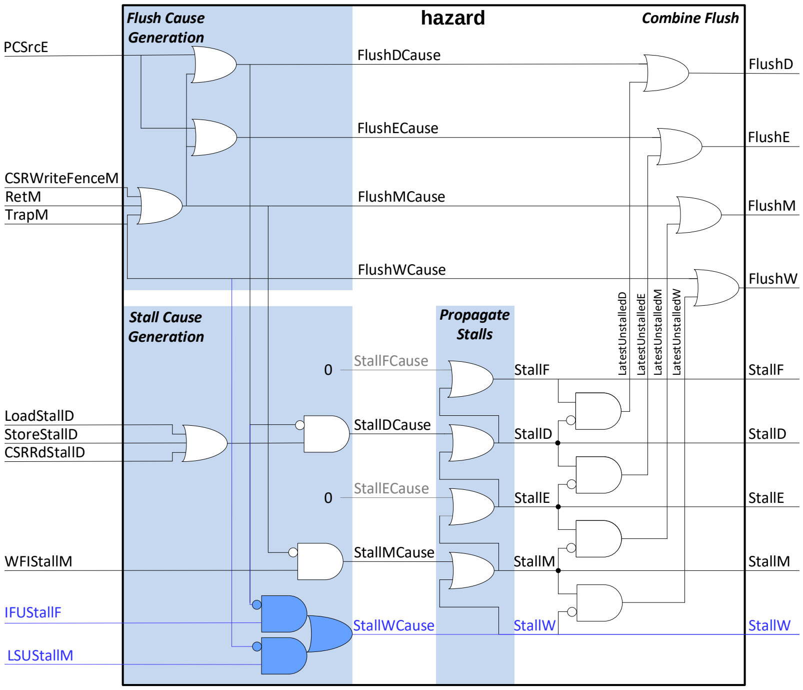


Figure 9.33 Hazard Unit, with bus changes highlighted in blue

When both the IFU and LSU need to access the bus, the EBU prioritizes the LSU. When the LSU completes its request and lowers `LSUStallM`, the pipeline will remain stalled until the IFU's request is finished. Because `StallW` is still asserted, the LSU's `busfsm` remains in Mem3 to prevent reissuing the bus request. After the IFU's `busfsm` transitions to Mem3, both units then lower their `BusStall` and the pipeline resumes.

<SIDEBAR>
`LSUStallM` is also asserted on cache and hardware page table walker stalls (Section 9.4.1XREF). Similarly, `IFUStallF` is also asserted on cache stalls (Section 10.3.5XREF) and spills (Section 11.3.2XREF).

<SIDEBAR>
Other hazards such as CSR writes may also stall the pipeline and keep the `busfsm` in Mem3 to prevent reissuing the bus request.
</END>

<SIDEBAR>
The Hazard Unit could choose to stall just the Decode stage when `IFUStallF` asserts, which would allow the rest of the pipeline to continue by inserting a bubble in the Decode stage. However, stalling the whole pipeline simplifies the implementation of traps. If the Decode stage is stalled and a load or store generates an exception, the IFU could be in the process of fetching the next instruction. The Hazard Unit would need to stall the memory stage until the IFU lowers `BusStall` to ensure the exception is taken. This problem is made more complex with the addition of caches in Chapters 9 and 10. Thus Wally stalls the whole pipeline during both IFU and LSU bus accesses.
<END>

The `busfsm's` `BusCommitted` signal delays pending interrupts until the bus transaction completes. `BusCommitted` must remain high while waiting in Mem3 so that the interrupt does not occur until the next instruction. `BusCommitted` is zero only during the Adr phase, which allows interrupts to occur before either the LSU or IFU starts a bus access. If a trap occurs while the IFU or LSU is in the Adr phase, the bus transaction is canceled by driving `HTRANS` to 0.

Figure 9.34 shows how the IFU `busfsm` delays interrupts until the current bus transaction completes. The CPU is fetching the `ori` instruction from 0x80001000 in Code Example 9.1 when the interrupt becomes pending at cycle 3. The `busfsm` asserts `BusCommittedF` when entering the Data ohase in cycle 2, which delays interrupts until the transaction completes. The bus completes fetching `ori` in cycle 5 and lowers `BusCommittedF` in cycle 6, which allows the pending interrupt to be handled. `TrapM` flushes the pipeline, including the `addi` fetch that was entering the Adr phase. In cycle 7, the trap handler begins fetching the `auipc` from 0x80002000, which initiates the Adr phase of the bus.

Code Example 9.1

```
80001000:      ori  x4, x0, 10
80001004:      addi x5, x4, 10
…
TrapHandler:
```

```
80002000:      auipc x6, 24
```

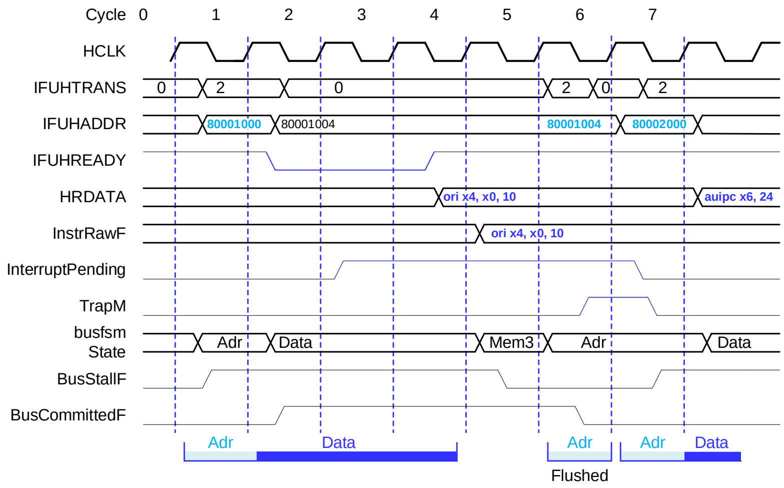


Figure 9.34 Privileged Unit interlock with the IFU's `busfsm`

<SIDEBAR>
The IFU could allow `TrapM` in cycle 5 to flush the completed fetch because the IFU only reads instructions. In contrast, once an instruction begins to write to memory, it must commit. Section 8.2.2.1XREF defines *nonidempotent* reads that have side effects and also must commit. For example, reading the Receive Buffer Register (RBR) in a 16550 UART also empties the buffer. Therefore, the RBR is nonidempotent because consecutive reads to the same address return different values.
</END>

The LSU similarly asserts `CommitedM` when its `BusCommitedM` is active. The `trap` module in the Privileged Unit asserts `Committed` when either the LSU `CommitedM` or IFU `CommitedF` are active. Committed delays interrupts until the instruction completes. Figure 9.35 shows the `trap` unit from Figure 5.9XREF with the new Committed logic highlighted in blue.

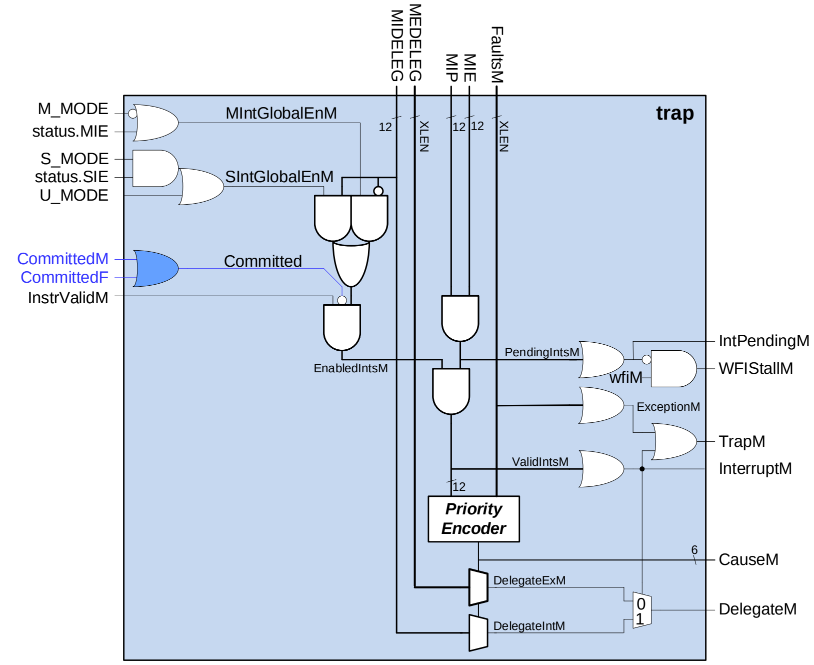


Figure 9.35 `trap` **unit delays pending interrupts while committed instructions complete a transaction**

Summary

Exercises