Interfacing Memory to the TMS320C32 DSP

Peter Galicki
Digital Signal Processing Solutions—Semiconductor Group

SPRA040A
June 1996

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Contents

List of Illustrations

List of Examples

ABSTRACT

The TMS320C32, a low-cost floating-point digital signal processor, makes the advanced 32-bit architecture of the TMS320C3x family available to a wider variety of applications than ever before. This application report explains the features of the ’C32 enhanced memory interface and gives examples of ways to interface external memory to the ’C32. These examples include interfacing to 32-, 16-, and 8-bit-wide external memories. Also discussed is generation of the $\overline{\text{RDY}}$ signal when a single strobe controls multiple external memory banks or peripherals, with some of them requiring wait states.

INTRODUCTION

The TMS320C32 digital signal processor is a low-cost member of the TMS320C3x generation of 32-bit floating-point processors. The features of the TMS320C32 reduce the chip and system costs and increase system performance, making the advanced 32-bit floating-point architecture of the ’C3x DSPs available to a wide spectrum of cost-sensitive applications. This application report explains the features of the TMS320C32 enhanced memory interface, and, by using design examples, it applies these features to illustrate the many possible ways to interface external memory to the ’C32. By emphasizing system solutions, this report also highlights the inherent flexibility of the memory interface to adapt efficiently to a variety of low-cost applications.

In addition to interfacing to 32-bit-wide memory, the ’C32 fully supports program fetches from 16-bit-wide memory and data access cycles from 16- and 8-bit-wide memory. Memory of any width (8, 16, or 32 bits) can be used to store data of any size (8, 16, or 32 bits); that is, memory width and data size need not be the same. Multiple strobes of the enhanced memory interface give the ’C32 the flexibility to directly access several memory banks of different widths and speeds. Up to three banks can be simultaneously interfaced without any glue logic. The ’C32’s ability to internally pack and unpack individual bytes of data can significantly reduce the external memory chip count to just one 8-bit-wide SRAM (with programs running internally).

Like the previous members of the ’C3x generation, the TMS320C32 is a 32-bit device with 32-bit internal memory, 32/40-bit internal registers, 32-bit internal buses, and other features. In fact, the 8/16-bit external memory interface can be completely transparent to the programmer after initializing two control registers. For example, an application running on the ’C32 can operate exclusively on 32-bit data types, but the external memory can be limited to one 8-bit-wide SRAM. In this case, the ’C32 processes the data just like the ’C30 or ’C31, but when it reads or stores the 32-bit words, it accesses the external memory in four cycles (one byte at a time) instead of one cycle.

As a different example, a ’C32 program can access integer data that never exceeds a value of 256. The enhanced memory interface can be programmed to convert the internal 32-bit words to external 8-bit words every time this data is accessed. The external 8-bit-wide memory is accessed in one cycle, and each 32-bit internal word is accessed as a single byte externally, resulting in memory savings of three bytes per 32-bit word.

OVERVIEW OF THE ENHANCED MEMORY INTERFACE

The ’C32 accesses external memory with one 24-bit address bus, one 32-bit data bus, and three strobes: IOSTRB, STRB0, and STRB1. The strobes are mapped to selected portions of the memory map as shown in Figure 1. For example, if the CPU is reading data from location 881234h, the active strobe during the read bus cycle is STRB0. Unlike the other two strobes, STRB0 is assigned to two noncontiguous address spaces within the memory map to provide extra flexibility in address decoding for glueless memory interfaces.

The behavior of IOSTRB is similar to its counterpart in the TMS320C30. Its timing characteristics are slightly relaxed in comparison with STRB0 and STRB1 cycles to better accommodate slower I/O peripherals. In contrast to STRB0 and STRB1, IOSTRB uses a single signal line and accesses the external data one full 32-bit word at a time. STRB0 and STRB1 are composed of four signal lines each. The multiple signal lines per strobe enable the STRB0 and STRB1 cycles to access external memory one byte, a half-word, or a full word at a time. For example, to read a single byte from a 32-bit-wide external memory location mapped to STRB0, the address on the address bus points to the selected 32-bit word and only one STRB0 signal is activated (driven low) to select the desired byte. To access two bytes of data at the memory location mapped to STRB1, two STRB1 signal lines are asserted during the bus cycle. Full 32-bit bus cycles involving STRB0 or STRB1 memory space result in four strobe signals simultaneously accessing four bytes of data. The 32-bit STRB0 and STRB1 bus cycles are no different functionally from the IOSTRB cycles but simply have tighter timing parameters.

The STRB0 and STRB1 cycles are not limited to just selecting bytes out of 32-bit memory locations. There are two strobe control registers that configure the data size and memory width for STRB0 and STRB1 bus cycles (one control register per strobe). With proper initialization of the strobe control registers, the bus cycles can be configured to encompass any combination of data size and physical memory widths. For example, a byte can be read from a 16-bit-wide memory or a 32-bit word can be written to an 8-bit-wide memory by configuring memory width and data size fields of the corresponding strobe control registers (see Figure 1).

As is the case with the other members of the ’C3x generation, the ’C32 program, as well as data, can reside in any portion of the memory map. The ’C32 program fetches from address space mapped to IOSTRB are indistinguishable from IOSTRB data reads or writes. However, the STRB0 and STRB1 cycles are configured slightly differently for program fetches than for data accesses. Program and data can still share the same portions of the memory map, but instead of setting the memory width and data size fields in STRB0 and STRB1 control registers, the program fetch cycles from the memory spaces mapped to STRB0 and STRB1 are configured by hardwiring the PRGW (program memory width select) pin. There is no need to use the data size fields, because all program fetches apply only to instruction words that are 32 bits wide. The memory width field of the strobe control register is useless at reset, when the processor is fetching the reset vector from memory. At that point the strobe control register is always configured in the same way, but different systems can have different memory widths. The PRGW pin indicates to the memory interface whether the program memory is 16 or 32 bits wide. Eight-bit program memory is not supported, because four cycles per instruction degrade the performance too much to be useful for most applications.

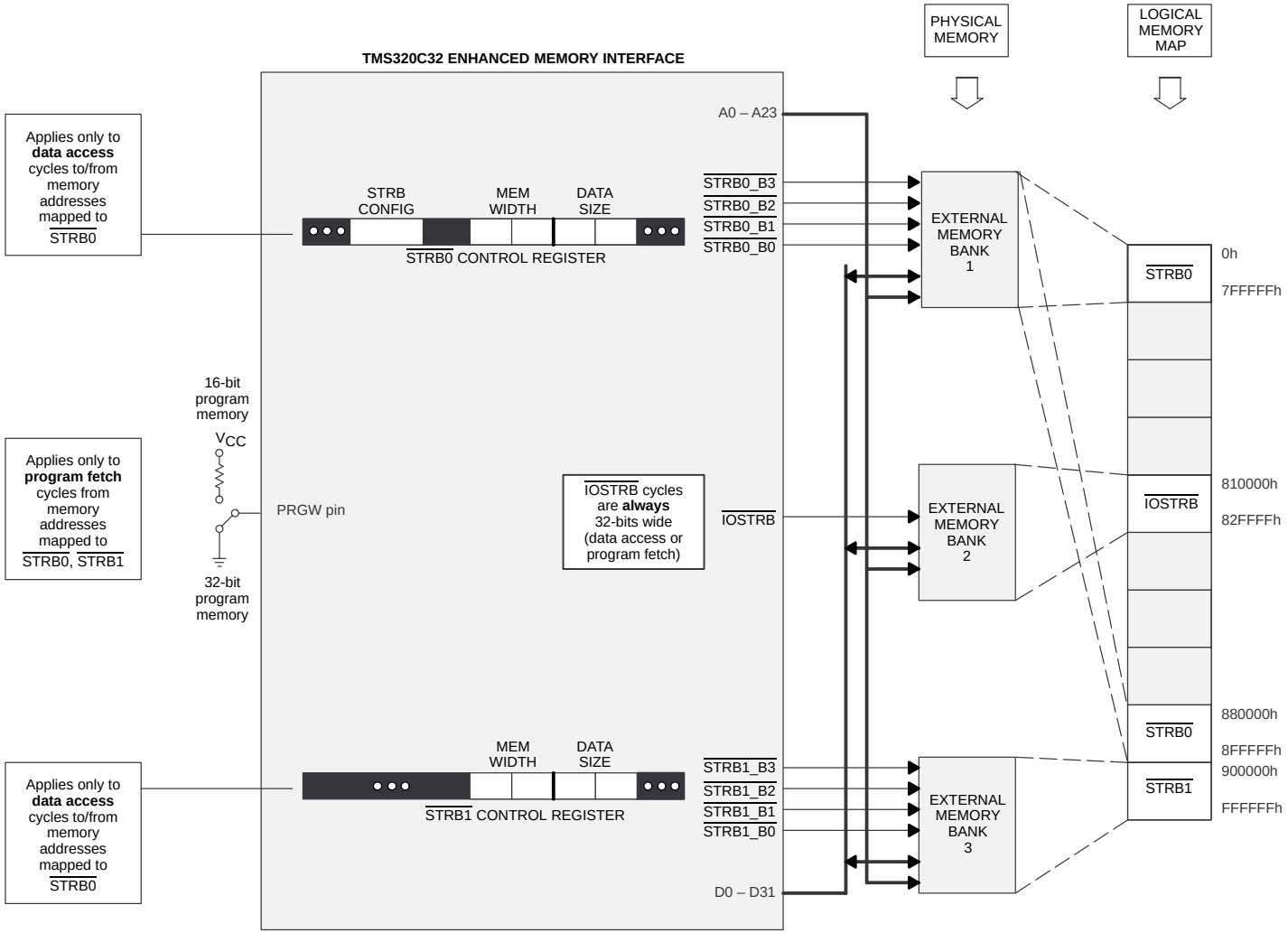


Figure 1. STRB0 and STRB1 Control Registers and the PRGW Pin

FUNCTIONAL DESCRIPTION OF THE ENHANCED MEMORY INTERFACE

As evident in Figure 1, Figure 2, Figure 3, and Figure 4, the enhanced memory interface controls all data and program traffic between data buses inside the chip and the 32-bit external memory bus. For any bus cycle involving a logical memory address range mapped to $\overline{\text{IOSTRB}}$, the memory interface simply connects the external data bus with an appropriate internal data bus without further data manipulation.

The memory interface is much busier when the ’C32 is accessing logical memory addresses mapped to $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$. Depending on the data size and external memory width (as defined by corresponding strobe control registers), data can be packed, unpacked, truncated, or shifted on its way to and from the chip.

The following examples illustrate how the data is manipulated when the interface has to match variable-size data with 8-, 16-, and 32-bit-wide physical memories. In these examples, five lines of code (included in each example’s program space) read five integers from one data space, convert them to floating-point format, and write them to another memory space that is assigned to a different strobe. Each example has a different combination of data sizes and external memory widths to illustrate the range of possible combinations.

For data access and program fetch cycles in which the data size exceeds the physical memory width, the least significant bytes/half-words are always transferred first.

Example 1. $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ Data Access: Data Size = Memory Width

This example illustrates a savings in external memory by using bytes and half-words to store data that is less than 32 bits in size (see Figure 2). As in all cases, the data size and memory width for $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ data access cycles are configured in the corresponding strobe control registers (see Table 1).

The short program stored in the internal RAM0 memory begins with the LDI instruction reading an 8-bit integer from 8-bit-wide $\overline{\text{STRB0}}$ memory. As the integer data passes through the memory interface, it is sign-extended to 32 bits and loaded to R0 as a 32-bit integer. Next, the FLOAT instruction converts the integer in R0 to a 40-bit floating-point number and loads it to R1. Finally, the STF instruction truncates the 40-bit contents of R1 to 32 bits and stores it in the 16-bit-wide $\overline{\text{STRB1}}$ memory. As the data passes through the memory interface, the 24-bit mantissa is truncated to eight bits (the 8-bit exponent remains unmodified).

Table 1. $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ Data Access: Data Size = Memory Width

	Strobe	Data Size	Memory Width
Input Data	$\overline{\text{STRB0}}$	8	8
Output Data	$\overline{\text{STRB1}}$	16	16
Program	RAM0	32	32

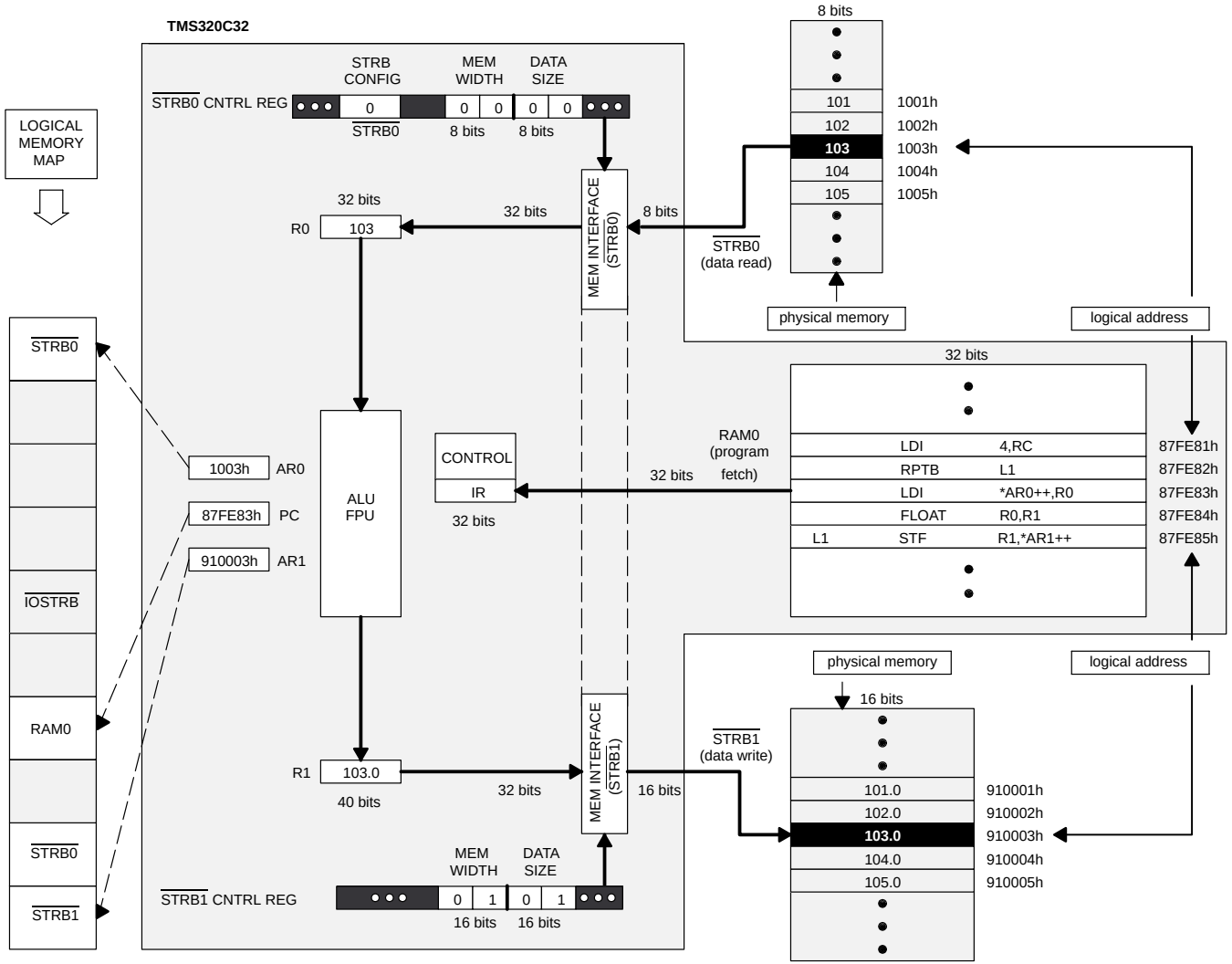


Figure 2. $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ Data Access: Data Size = Memory Width

Example 2. $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ Data Access: Data Size ≠ Memory Width

This example shows that the input/output data does not have to be the same size as the memory from which it is being read or to which it is being written (see Table 2). As in all cases, the data size and memory width for $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ data access cycles are configured in the corresponding strobe control registers.

The short program stored in the RAM1 memory begins with the LDI instruction reading an 8-bit integer from 16-bit-wide $\overline{\text{STRB0}}$ memory (see Figure 3). Since each address contains two data bytes, the memory interface uses different $\overline{\text{STRB0}}$ lines to differentiate between the high byte and the low byte (both $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ comprise four signals each, one for each byte of the 32 bits). Next, the FLOAT instruction converts the integer in R0 to a 40-bit floating-point number and loads it to R1. Finally, the STF instruction stores the contents of R1 to 16-bit-wide memory as a 32-bit number. Before the data arrives at the memory interface, the 32-bit mantissa is truncated to 24 bits (the 8-bit exponent remains unmodified). The memory interface then stores the 24-bit mantissa and the 8-bit exponent in 16-bit-wide memory, two bytes at a time, using two cycles and two physical memory addresses.

Table 2. $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ Data Access: Data Size ≠ Memory Width

	Strobe	Data Size	Memory Width
Input Data	$\overline{\text{STRB0}}$	8	16
Output Data	$\overline{\text{STRB1}}$	32	16
Program	RAM1	32	32

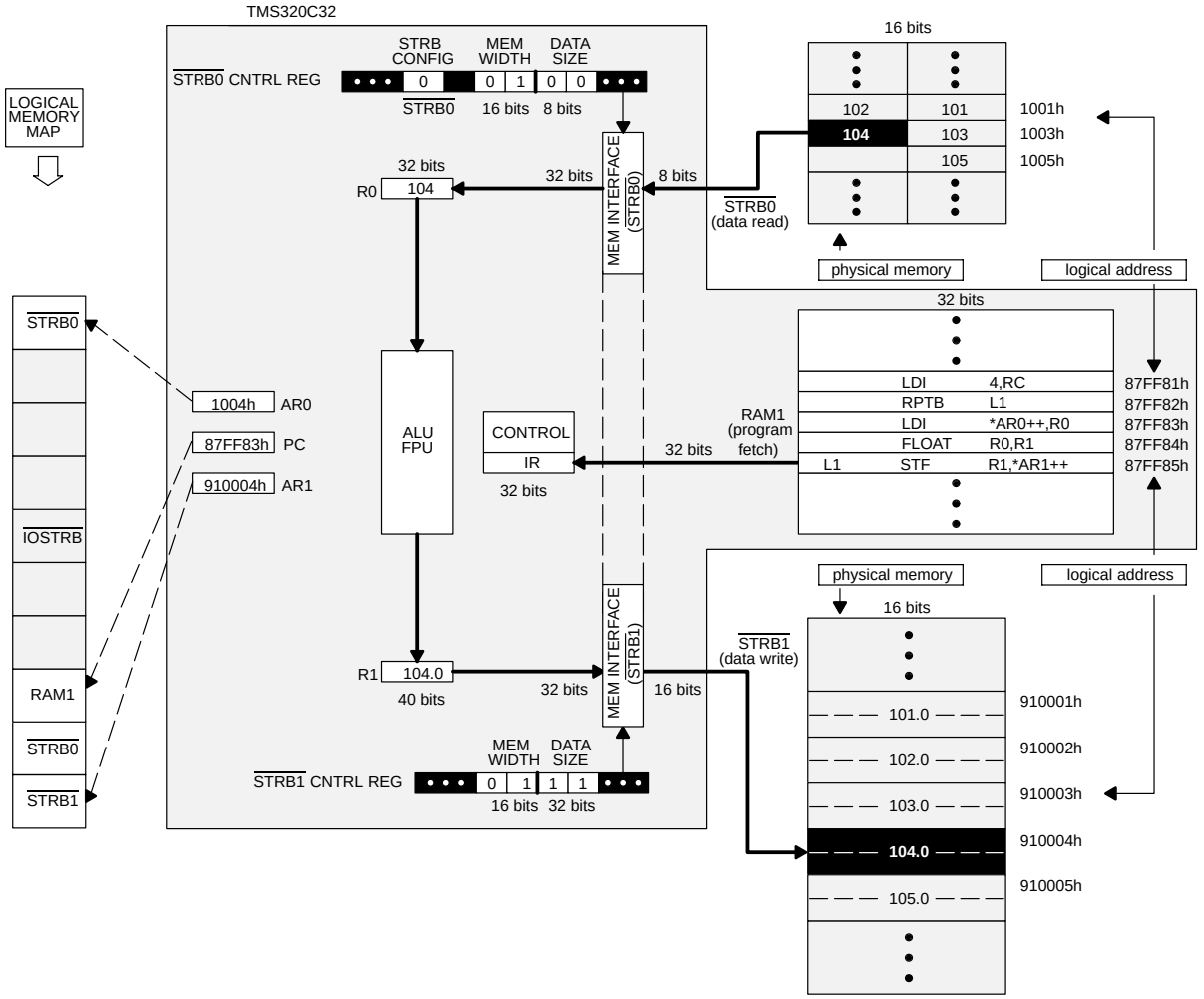


Figure 3. $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ Data Access: Data Size ≠ Memory Width

Example 3. Program Fetch From 16-Bit $\overline{\text{STRB0}}$ Memory

This example shows that program memory mapped to $\overline{\text{STRB0}}$ or $\overline{\text{STRB1}}$ space can be configured to 16 bits by hardwiring the PRGW pin to a high state (see Table 3). It also demonstrates that 32-bit data transfers to and from the 32-bit-wide external memory do not involve any data operations in the memory interface.

The short program stored in $\overline{\text{STRB0}}$ memory begins with the LDI instruction reading a 32-bit integer from 32-bit-wide $\overline{\text{IOSTRB}}$ memory and loading it to R0 (see Figure 4). Next, the FLOAT instruction converts the integer in R0 to a 40-bit floating-point number and loads it into R1. Finally, the STF instruction truncates the 40-bit contents of R1 to 32 bits and stores it in the 32-bit-wide $\overline{\text{STRB1}}$ memory. The data is not modified as it passes through the memory interface.

The program controlling the data conversion in this example is stored in the 32-bit-wide memory bank mapped to $\overline{\text{STRB0}}$. As discussed earlier, program fetch cycles do not reference the strobe control register to determine the width of the program memory. Instead, the memory interface checks the state of the PRGW pin to determine the memory width. Because the program memory is 16 bits wide, the PRGW pin should be pulled up to V_{CC}, effectively directing the memory interface to fetch instructions in two bus cycles per instruction (16 bits at a time).

Table 3. Program Fetch From 16-Bit $\overline{\text{STRB0}}$ Memory

	Strobe	Data Size	Memory Width
Input Data	$\overline{\text{STRB0}}$	32	32
Output Data	$\overline{\text{STRB1}}$	32	32
Program	$\overline{\text{IOSTRB}}$	32	16

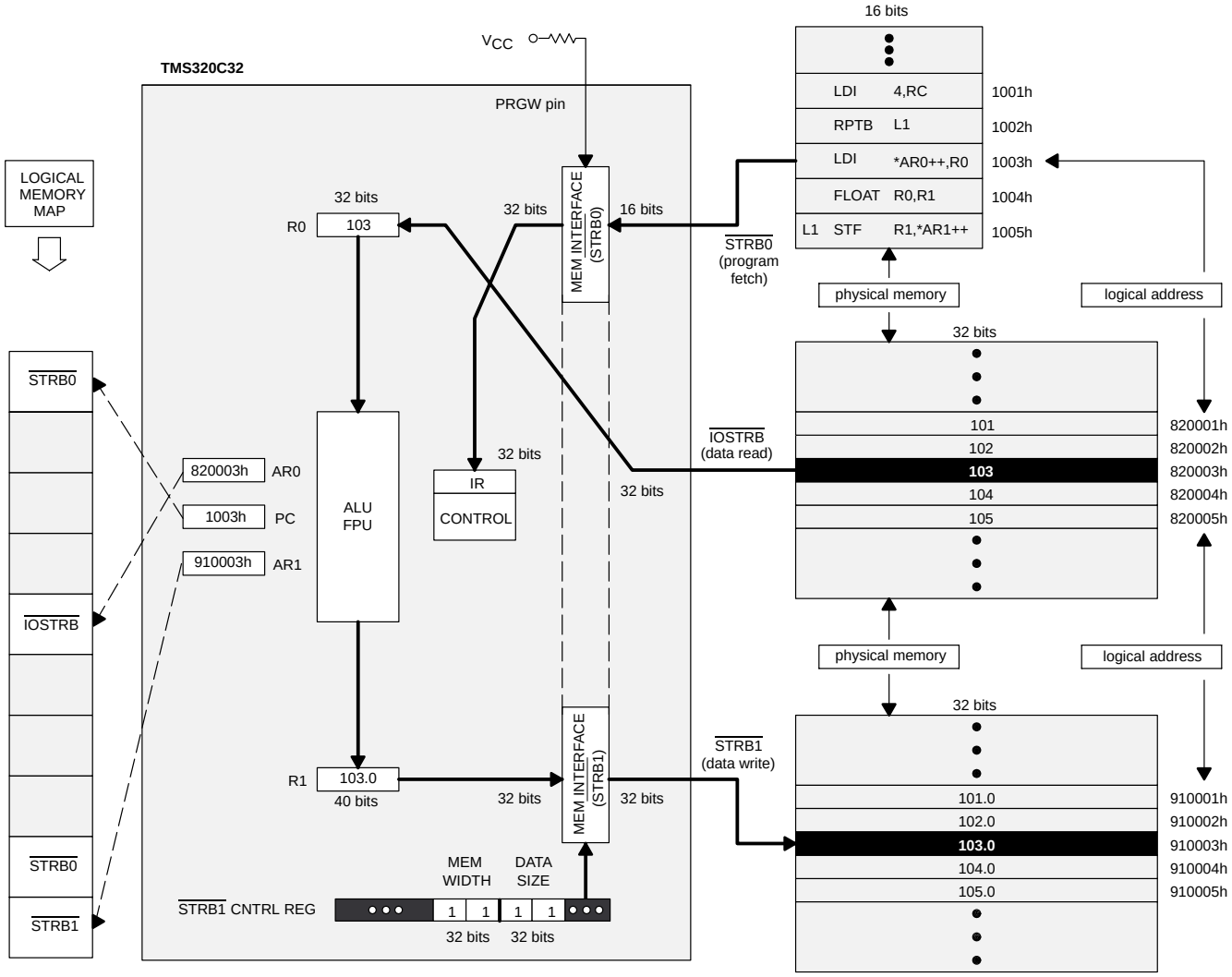


Figure 4. Program Fetch From 16-Bit $\overline{\text{STRB0}}$ Memory

Example 4. Program Fetch From 32-Bit $\overline{\text{STRB1}}$ Memory

This example shows that program memory mapped to $\overline{\text{STRB0}}$ or $\overline{\text{STRB1}}$ space can be configured to 32 bits by hardwiring the PRGW pin to a low state (see Table 4). It also demonstrates that 32-bit data transfers to and from the 32-bit-wide external memory do not involve any data operations in the memory interface.

The small program stored in $\overline{\text{STRB1}}$ memory begins with the LDI instruction reading a 32-bit integer from 32-bit-wide $\overline{\text{STRB0}}$ memory and loading it to R0 (see Figure 5). Next, the FLOAT instruction converts the integer in R0 to a 40-bit floating-point number and loads it into R1. Finally, the STF instruction truncates the 40-bit contents of R1 to 32 bits and stores it in the 32-bit-wide $\overline{\text{IOSTRB}}$ memory. The data is not modified as it passes through the memory interface.

The program controlling the data conversion in this example is stored in the 32-bit-wide memory bank mapped to $\overline{\text{STRB1}}$. As discussed earlier, program fetch cycles do not reference the strobe control register to determine the width of the program memory. Instead, the memory interface checks the state of the PRGW pin to determine the memory width. Because the program memory is 32 bits wide, the PRGW pin should be grounded, effectively directing the memory interface to fetch instructions in one bus cycle per instruction (32 bits at a time).

Table 4. Program Fetch From 32-Bit $\overline{\text{STRB1}}$ Memory

	Strobe	Data Size	Memory Width
Input Data	$\overline{\text{STRB0}}$	32	32
Output Data	$\overline{\text{STRB1}}$	32	32
Program	$\overline{\text{IOSTRB}}$	32	32

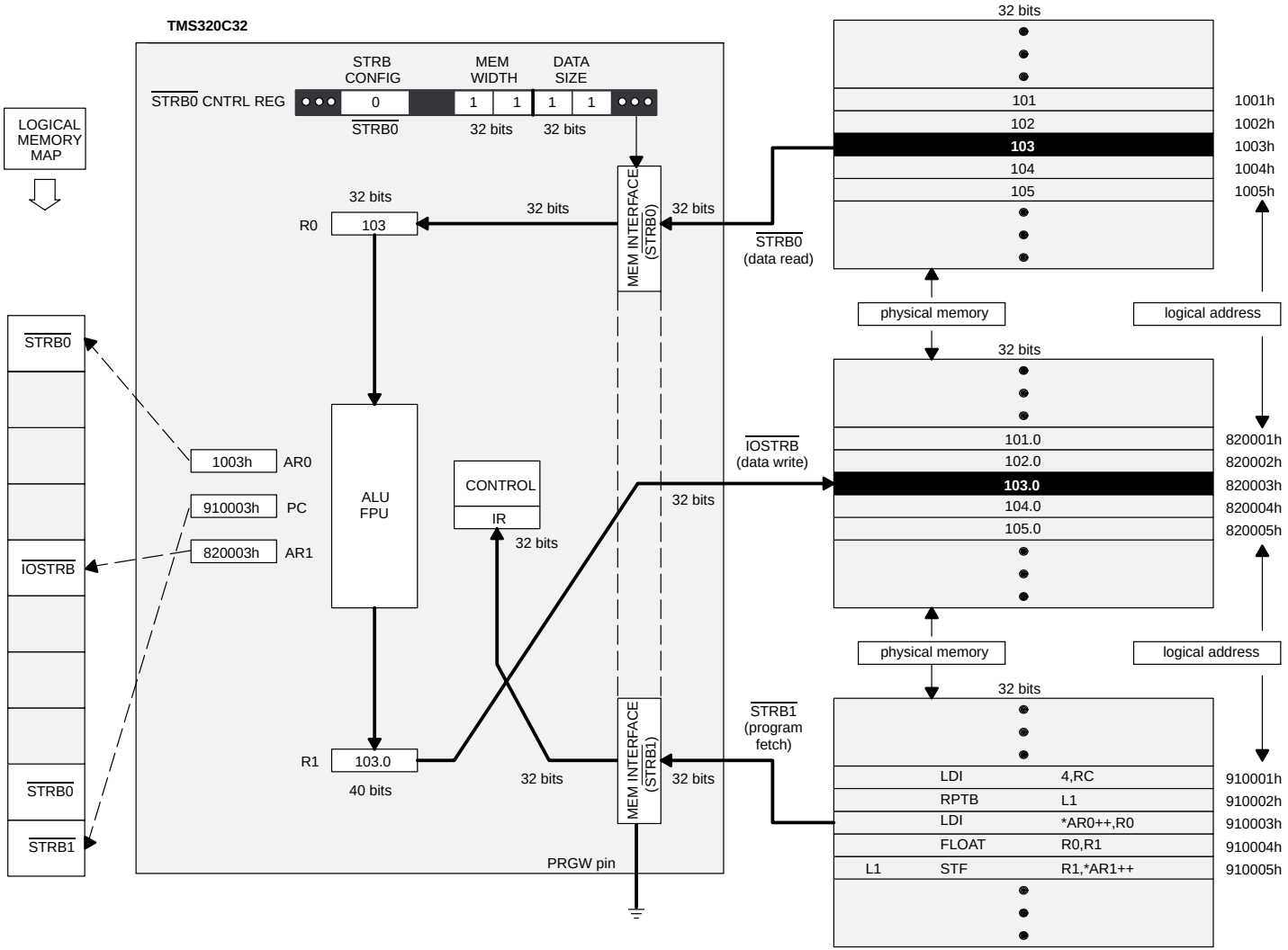


Figure 5. Program Fetch From 32-Bit $\overline{\text{STRB1}}$ Memory

LOGICAL VERSUS PHYSICAL ADDRESS

The ’C32 is a 32-bit processor. Its instruction set operates on 32-bit registers, and the CPU alone does not understand 8- or 16-bit data or data transfers. When a ’C32 instruction writes to a physical address, it sends all 32 bits of data to the memory interface unit via an internal bus. It is only in the memory interface that the internal 32-bit data can assume 8-bit or 16-bit form, provided that the address is in the $\overline{\text{STRB0}}$ or $\overline{\text{STRB1}}$ range of the memory map. The data size field of the $\overline{\text{STRB0}}$ or $\overline{\text{STRB1}}$ control register determines the actual size of the data portion that is placed on the external memory bus of the ’C32. Likewise, when a ’C32 instruction reads a portion of data from external memory, the memory interface always converts it to 32 bits as it enters the chip. What happens to the external data as it goes through the memory interface on the way to the CPU depends on the contents of the $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ control registers. Once again, only the data whose address falls within the $\overline{\text{STRB0}}$ or $\overline{\text{STRB1}}$ range of the memory map can be manipulated inside the memory interface unit.

Throughout this document, the term *logical address* applies to a memory location as it is referenced by ’C32 instructions and that is a part of the processor’s logical memory map. The *physical address* refers to the address that appears at the ’C32 address pins. The valid ranges of the logical memory map that the program instructions can reference are determined by the external memory available in the system, how the external memory address pins are matched with the ’C32 address pins (which depends on physical memory width), and the contents of the $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ registers (which define physical memory width and the data size).

The logical memory map shown in Figure 6 always contains 32-bit data as far as the CPU is concerned. It is only when the data passes through the memory-interface block that the data size can actually change to 8 or 16 bits, as directed by the appropriate strobe control register. For example, when the processor reads a byte (eight bits) from external memory, the 8-bit data is sign-extended or padded with 0s as it passes through the memory interface so that it becomes 32-bit data inside the ’C32. Likewise, when the processor writes the contents of a 32-bit register to 16-bit-wide external memory, the internal 32-bit data is truncated to 16 bits as it passes through the memory interface. The dashed lines inside the logical memory map in Figure 6 show the internal 32-bit representation of the external data that has a physical size of 8 or 16 bits.

Figure 6 also has additional explanation of logical/physical addresses and other terms related to the ’C32 memory interface.

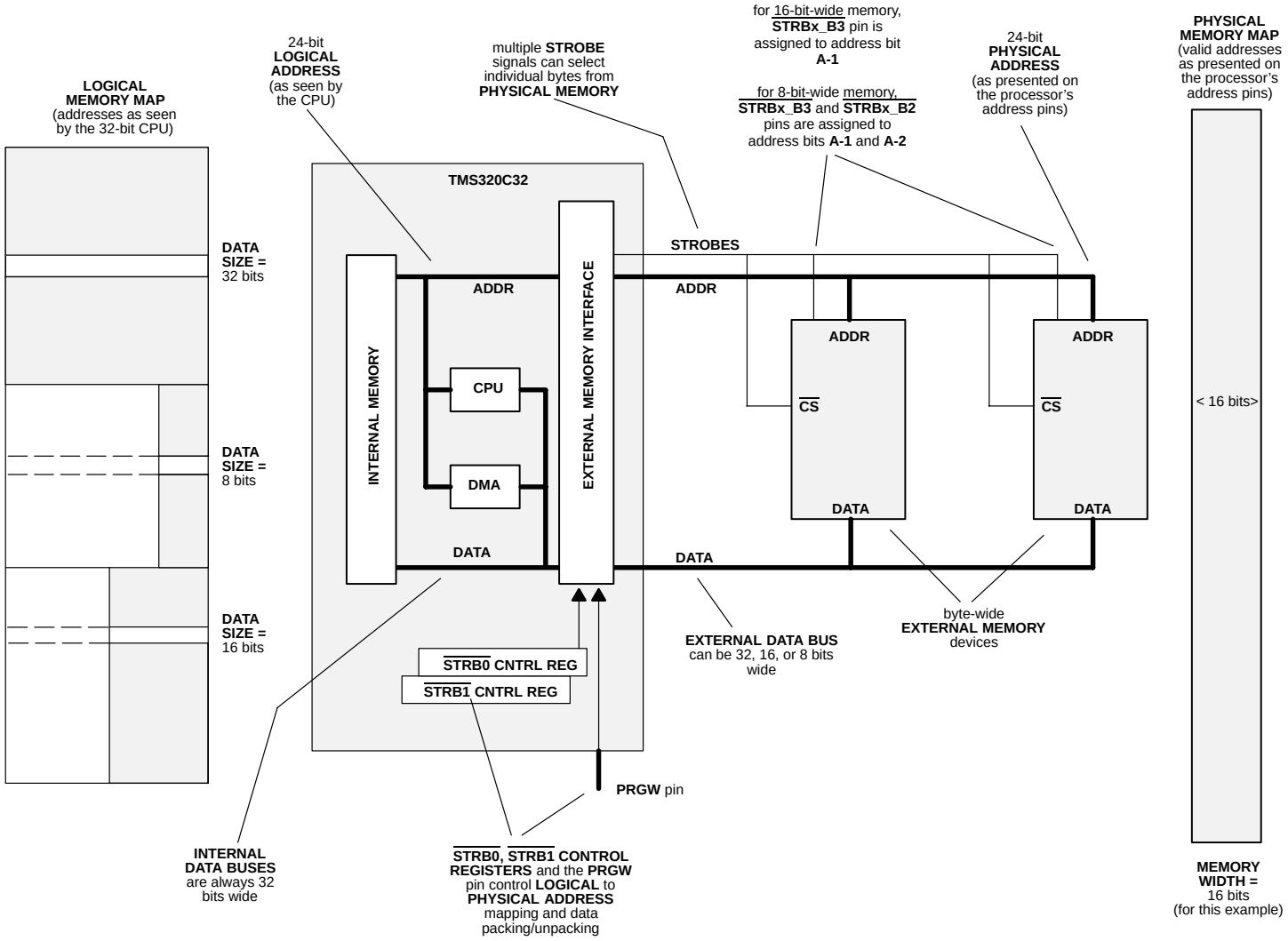


Figure 6. Description of Terms Involved in TMS320C32 Memory Interface

32-BIT MEMORY CONFIGURATION DESIGN EXAMPLES

The following two examples describe interfacing the ’C32 to 32-bit-wide external memory from both the hardware and software-addressing viewpoints.

Example 5. 32-Bit Memory Address Translation for Data Size = Memory Width

This example demonstrates that when both data size and memory width are 32 bits, the $\overline{\text{STRB0}}$ memory interface has the same functionality as that for $\overline{\text{IOSTRB}}$. The only difference between the two is the number of strobe lines connected to the respective memory banks: four for $\overline{\text{STRB0}}$ and one for $\overline{\text{IOSTRB}}$.

Figure 7 is a schematic diagram of a 32-bit interface consisting of two memory banks, each controlled by a separate strobe. The four signal lines of $\overline{\text{STRB0}}$ are assigned to the chip-select pins of four 32K × 8 15-ns SRAMs, and the single $\overline{\text{IOSTRB}}$ signal line is connected to the chip-enable pins of four 32K × 8 30-ns EPROMs. For the 60-MHz version of the ’C32, the 15-ns SRAMs operate with zero wait states and the 30-ns EPROMs require one wait state (software wait states can be programmed in the strobe control registers).

Figure 8 illustrates the programmer’s view of the hardware memory configuration depicted in Figure 7. The logical addresses (appearing in program instructions) are represented in the context of the entire memory map to identify the respective strobes. The physical addresses are the values that actually appear at the pins of the processor. Since $\overline{\text{IOSTRB}}$ operates exclusively on 32-bit data types, the memory interface does not modify the address going in and out of the CPU, and the logical and physical addresses are identical. In this example, $\overline{\text{STRB0}}$ also operates on 32-bit data since the memory width field of the $\overline{\text{STRB0}}$ control register contains a binary value of 11. Since the $\overline{\text{STRB0}}$ physical memory width is also 32 (see data size field), there is no need for any address translation from the logical address to its physical representation.

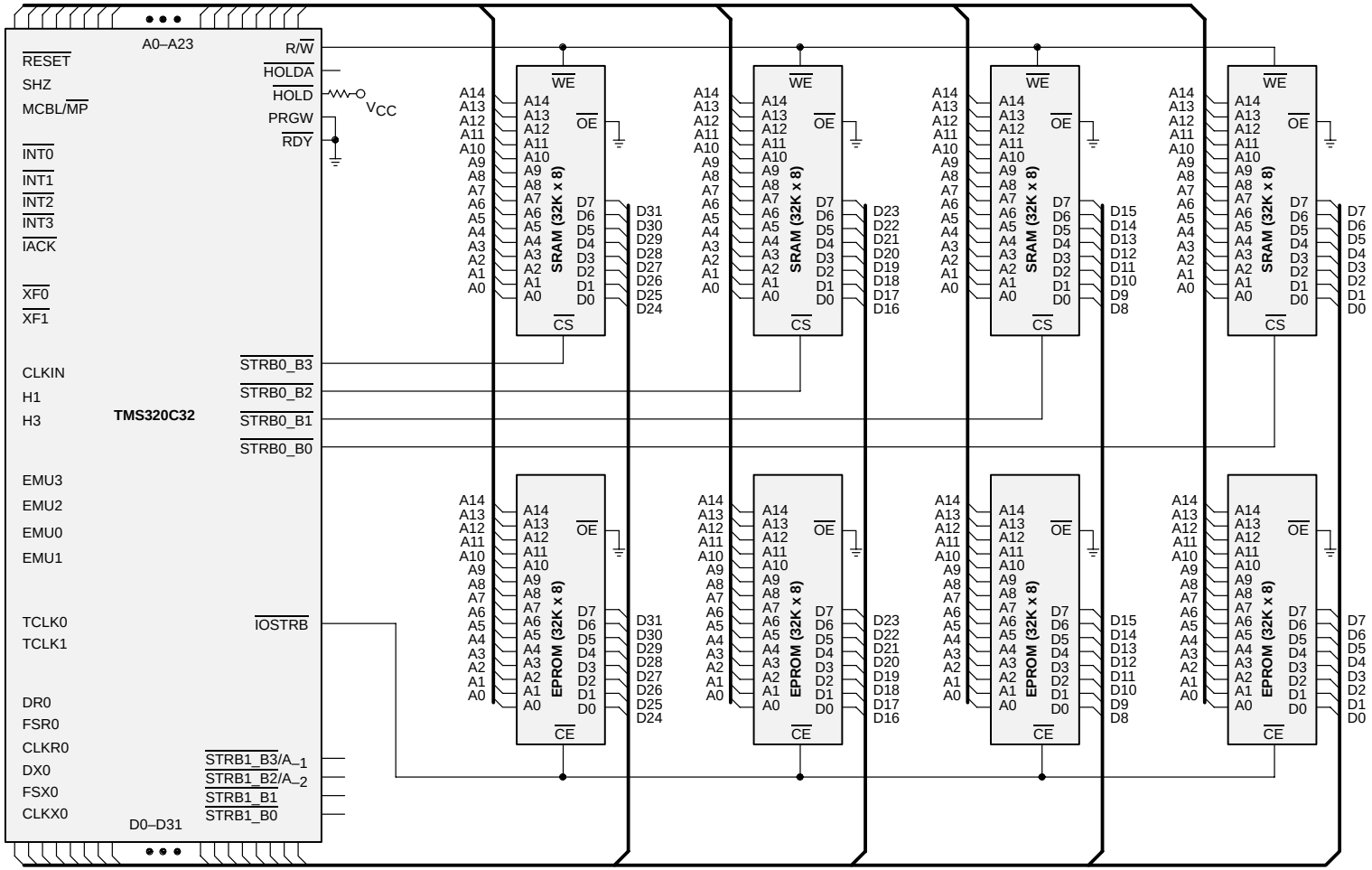


Figure 7. 32-Bit Memory Configuration ($\overline{\text{STRB0}}$ and $\overline{\text{IOSTRB}}$)

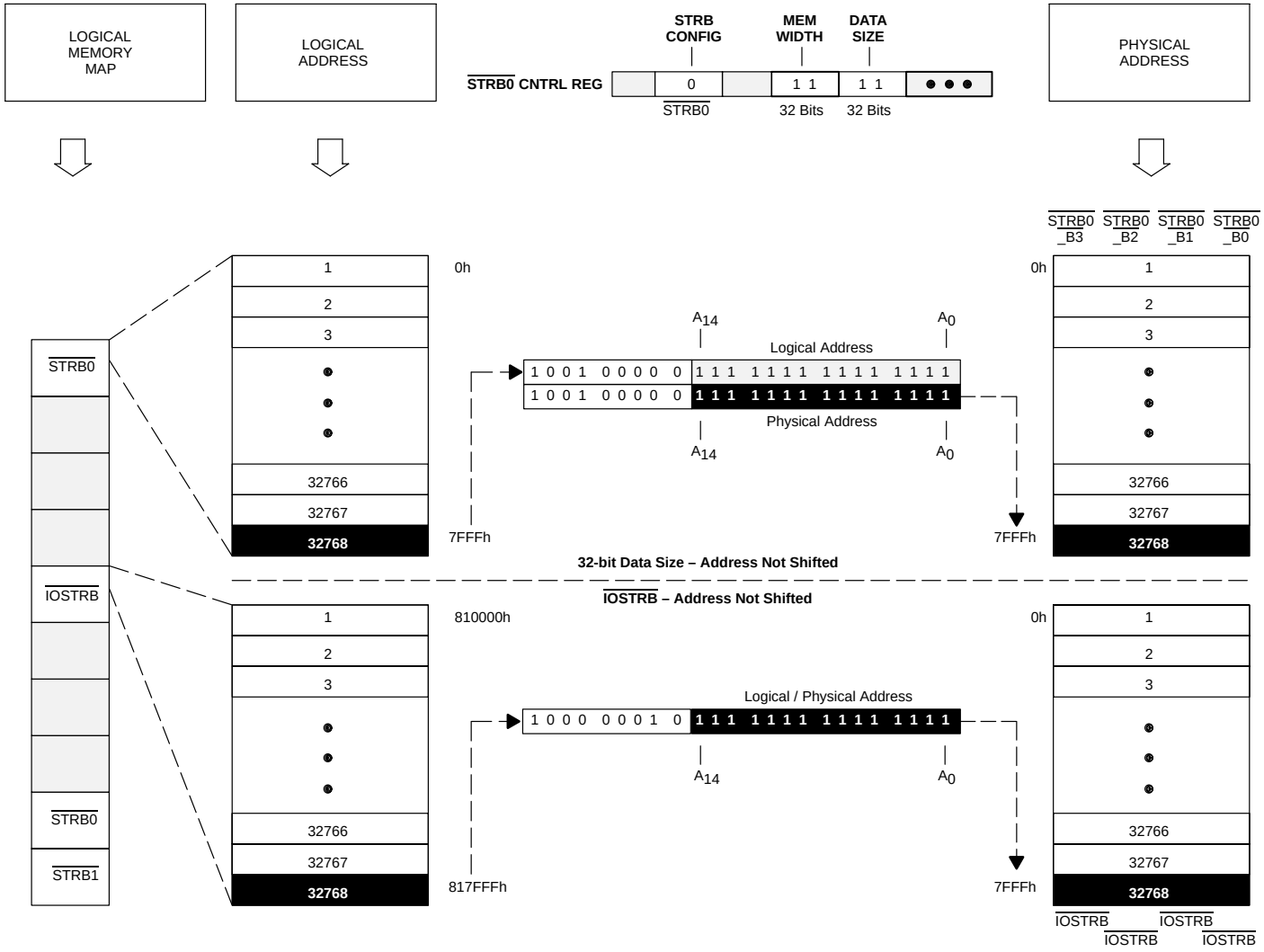


Figure 8. 32-Bit Memory Address Translation: Data Size = Memory Width

Example 6. 32-Bit Memory Address Translation for Data Size < Memory Width

This example demonstrates that if the data can fit in 16 or 8 bits of precision, the effective addressing range of the same physical 32-bit memory can be doubled or quadrupled by simply changing the data size field of the appropriate strobe control register before the transfers begin. The logical-to-physical address translation involves a 2-bit address shift if the data size is 8 bits and a 1-bit shift if the data size is 16 bits. Address shifts and the activation of selected external memory bytes with appropriate strobe control lines are automatically performed by the memory interface (as directed by strobe control registers) and can be considered transparent to the programmer.

Figure 9 is the schematic diagram of a 32-bit interface consisting of two memory banks, each controlled by a separate strobe. The four signal lines of $\overline{\text{STRB0}}$ are assigned to the chip-select pins of four 32K × 8 15-ns SRAMs, and the four signal lines of $\overline{\text{STRB1}}$ are connected to the chip-enable pins of four 32K × 8 30-ns EPROMs. For the 60-MHz version of the ’C32, the 15-ns SRAMs operate at zero wait states and the 30-ns EPROMs require one wait state (software wait states can be programmed in strobe control registers).

Figure 10 illustrates the programmer’s view of the hardware memory configuration depicted in Figure 9. The logical addresses (appearing in program instructions) are represented in the context of the entire memory map to identify the respective strobes. In this case, the $\overline{\text{STRB0}}$ memory transfers operate on 16-bit data to and from 32-bit-wide memory, as defined in the $\overline{\text{STRB0}}$ control register. $\overline{\text{STRB1}}$ accesses 8-bit data to and from 32-bit-wide memory, as defined by the $\overline{\text{STRB1}}$ control register. Since two 16-bit data types can fit in a single 32-bit-wide memory location referenced by a single physical address, a mechanism is needed to distinguish between the 16-bit data portions. This is accomplished by using the least significant bit of the logical address to activate a different pair of the four $\overline{\text{STRB0}}$ signal lines for each access, leaving the second least significant bit of the logical address to become the least significant bit of the physical address and effectively shifting the logical address by one bit. Similarly, $\overline{\text{STRB1}}$ 8-bit data transfers to the 32-bit-wide external memory cause the address to be shifted by two bits, because the two least significant bits of the logical address are used to select one out of four bytes sharing the same physical 32-bit memory location.

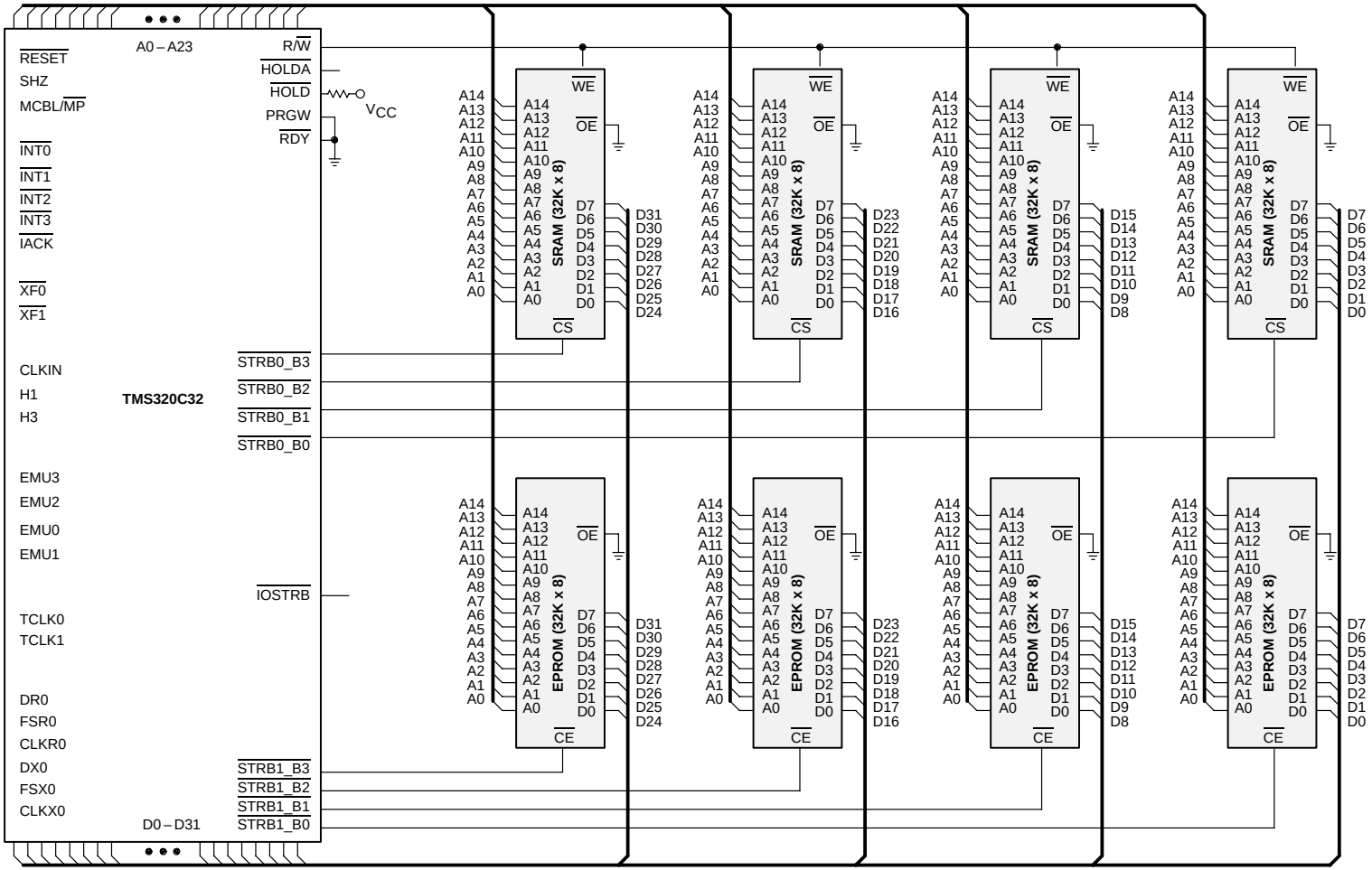


Figure 9. 32-Bit Memory Configuration ($\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$)

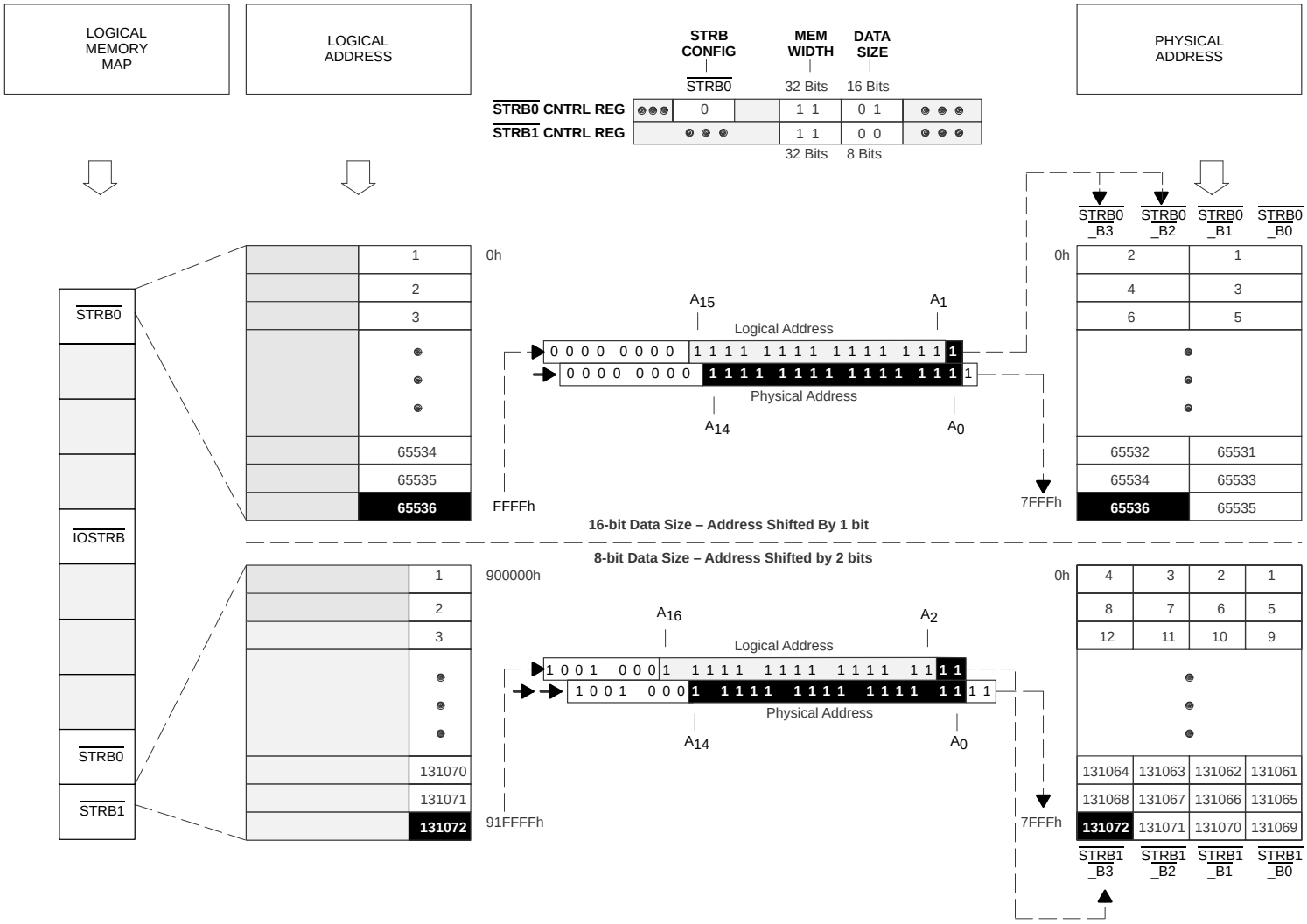


Figure 10. 32-Bit Memory Address Translation: Data Size < Memory Width

16/8-BIT MEMORY CONFIGURATION DESIGN EXAMPLES

The following examples describe from both the hardware and software-addressing perspectives how to interface the ’C32 to both 8- and 16-bit-wide external memories in the same design.

Figure 11 contains a schematic diagram of the external memory interface consisting of two banks, each controlled by a separate strobe. Two of four $\overline{\text{STRB0}}$ signal lines are assigned to the chip-select pins of two 32K $\times$ 8 15-ns SRAMs, and one of four $\overline{\text{STRB1}}$ signals is connected to a chip-enable pin of one 32K $\times$ 8 30-ns EPROM. For the 60-MHz version of the ’C32, the 15-ns SRAMs operate at zero wait states and the 30-ns EPROMs require one wait state (software wait states can be programmed in strobe control registers). Any time the external memory is less than 32 bits wide, some of the strobe pins switch functions and become additional address pins. For 16-bit-wide memory, $\overline{\text{STRB0\_B3}}$ becomes A_{-1}, and for 8-bit-wide memory, $\overline{\text{STRB1\_B3}}$ and $\overline{\text{STRB1\_B2}}$ become A_{-1} and A_{-2}, respectively. This is the only external change that differentiates the 32-bit-wide memory interface from the 16- and 8-bit-wide memory interfaces. This feature can be considered transparent to the software programmer, except that the programmer must configure the strobe control registers appropriately. The memory interface automatically drives the additional address lines with correct values, depending on the size of the data being transferred.

The following three examples illustrate how the physical addresses are derived from the logical addresses when the data size is equal to, greater than, and less than the width of the physical memory. Though address translation is completely automatic, these cases provide insight into the range of physical addresses actually affected during transfer of 32-, 16-, and 8-bit data.

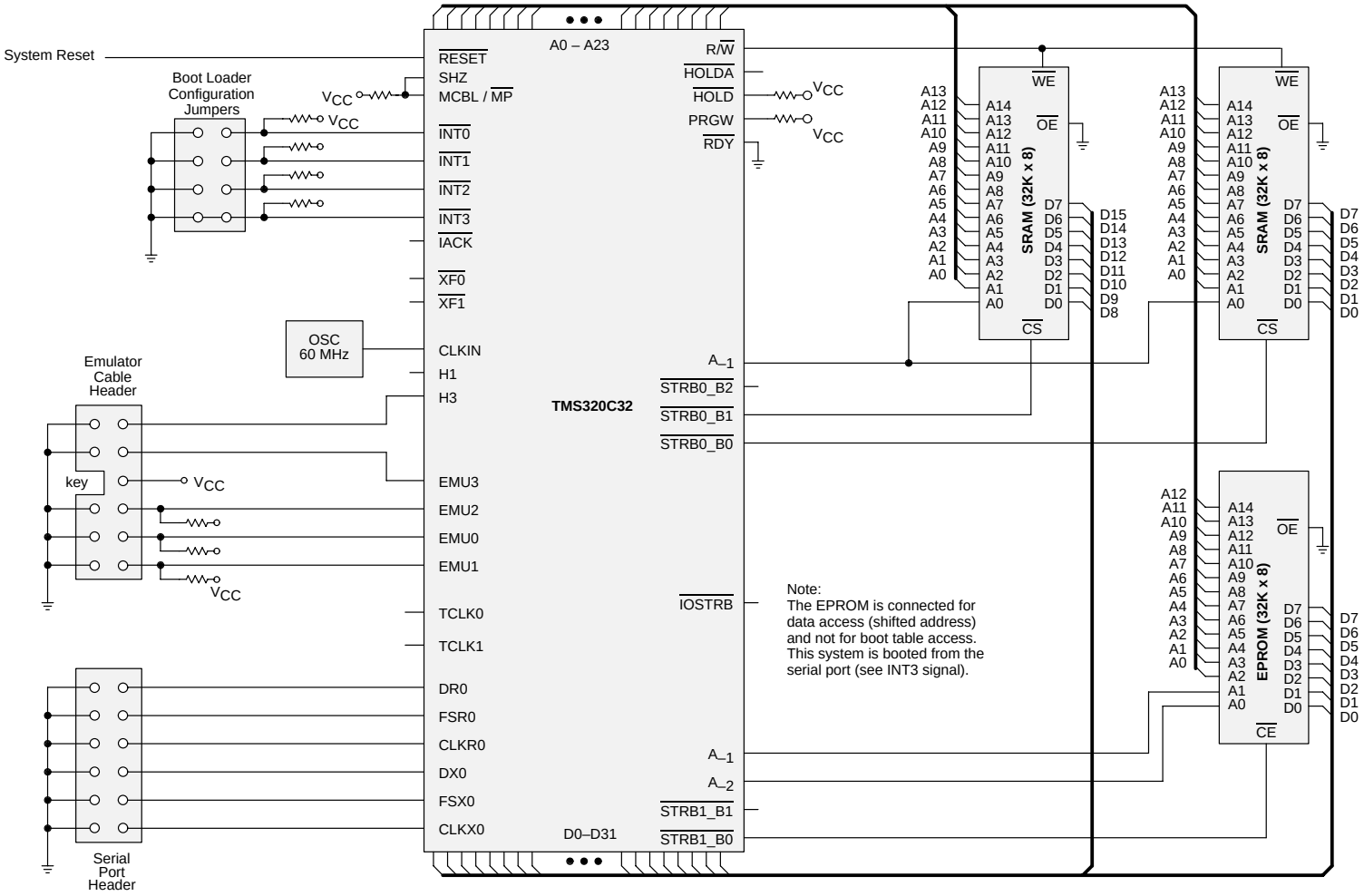


Figure 11. 16/8-Bit Memory Configuration: A Complete Minimum Design

Example 7. 16/8-Bit Memory Address Translation for Data Size = Memory Width

As shown in Figure 12, when the external memory width matches the size of data being transferred, the physical address also matches the logical address with one exception: the physical address is shifted relative to the logical address by one bit for 16-bit transfers and by two bits for 8-bit transfers. This means that the address bit that would normally be expected on pin A0 actually appears on pin A_{-1} or A_{-2}. As Figure 12 shows, there is one-to-one correspondence between logical data and its counterpart in physical memory.

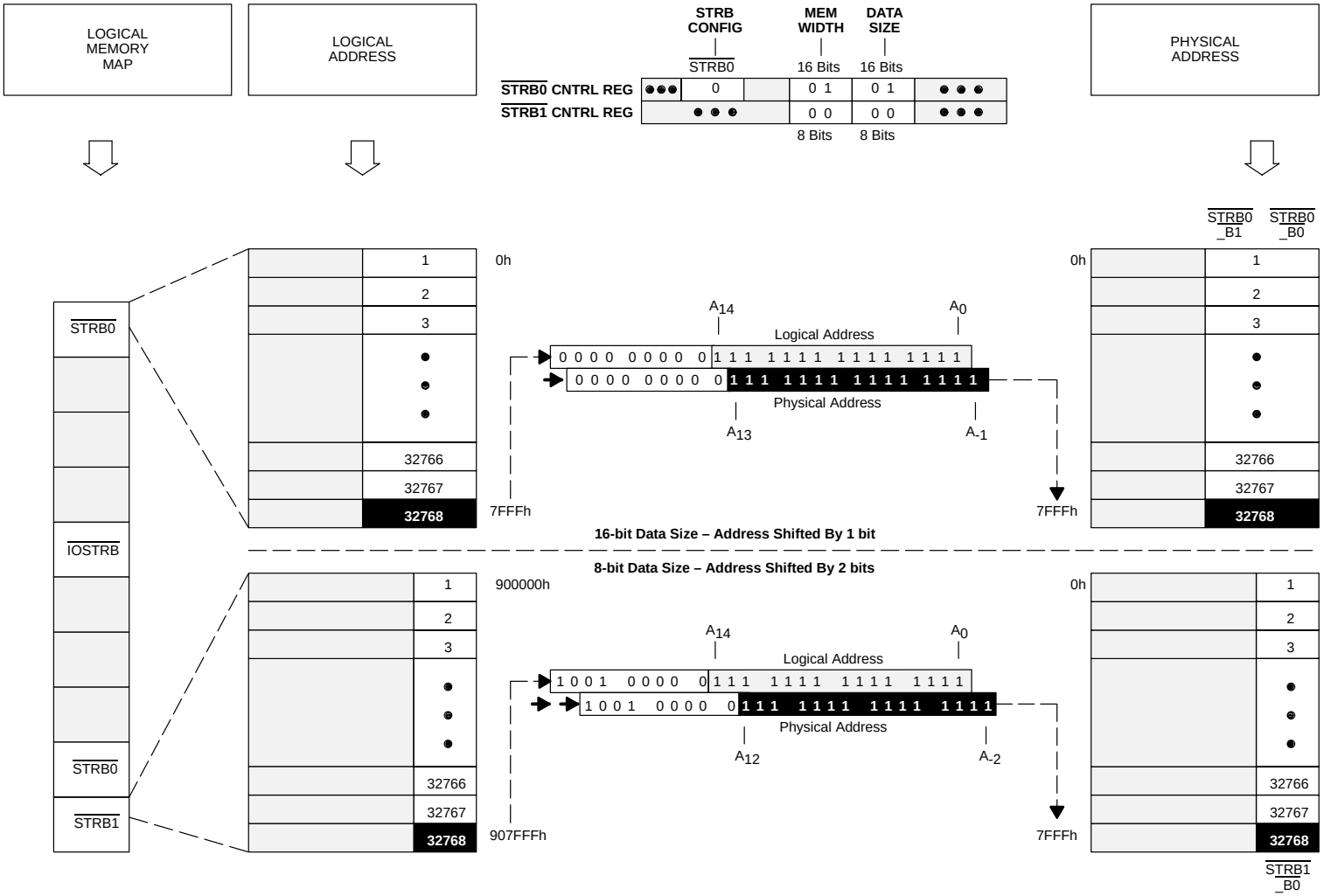


Figure 12. 16/8-Bit Memory Address Translation: Data Size = Memory Width

Example 8. 16/8-Bit Memory Address Translation for Data Size > Memory Width

Figure 13 depicts what happens when data that is larger than the physical memory in which it is to reside is transferred. As shown by the contents of the strobe control registers, $\overline{\text{STRB0}}$ controls transfers of 32-bit data to/from 16-bit-wide physical memory and $\overline{\text{STRB1}}$ controls transfers of 16-bit data to and from a byte-wide memory. When an instruction stores 32-bit data to logical address 0h, the memory interface must perform two write cycles to 16-bit-wide external memory. These two write cycles involve two consecutive addresses, 0h and 1h. A 16-bit portion of data logically referenced with a single address actually takes two physical addresses to store in 8-bit-wide physical memory (as is the case with the $\overline{\text{STRB1}}$ transfer shown at the bottom of Figure 13). To implement these extra bus cycles, the memory interface appends an extra address bit to the least significant end of both addresses. As in Example 7, the least significant bits of the $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ addresses appear at pins A_{-1} and A_{-2}, respectively, because they represent 16- and 8-bit-wide memories.

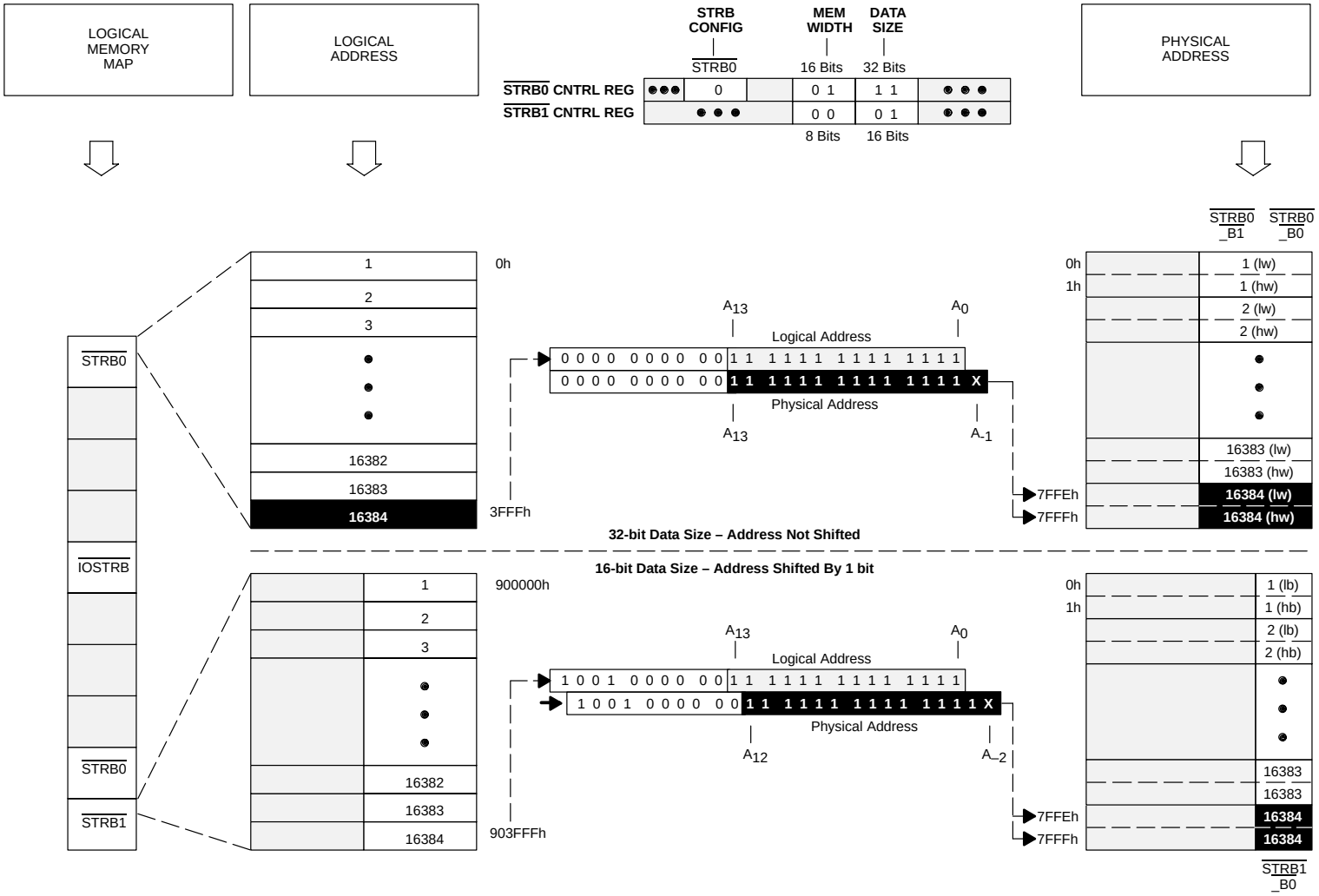


Figure 13. 16/8-Bit Memory Address Translation: Data Size > Memory Width

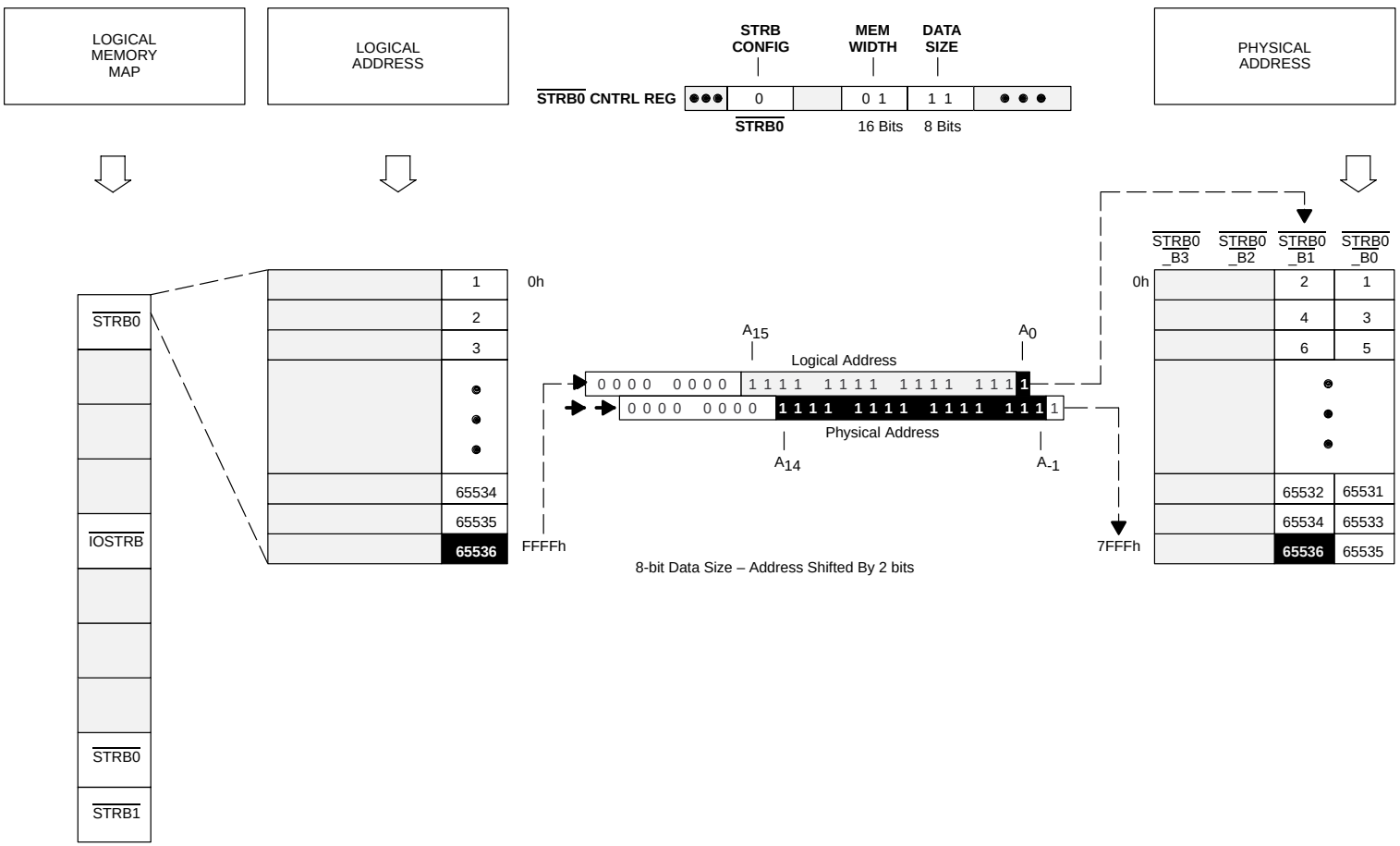
Example 9. 16/8-Bit Memory Address Translation for Data Size < Memory Width

The example in Figure 14 is, in a way, an inverse of Example 8. The 8-bit data is transferred to/from 16-bit-wide external memory. To put this example in perspective, assume that the data transfer is triggered by the following ’C32 instruction: STI R0,@7FFFh. While in R0, the data is sized at 32 bits, but when it arrives at the memory interface, the STRB0 control register data size field indicates 8-bit-wide data. So, the 32-bit data is truncated to eight bits. The next stop for the now byte-sized data is address 7FFFh of the 16-bit-wide external memory. Should it fill the high or low portion of that memory address? In this case, the LSB of the logical address (as referenced by the instruction) is actually rerouted to control one of the two STRB0 lines assigned to the 16-bit physical memory. If the LSB is 1 (as in this case), STRB0_B1 is asserted during the write cycle. If the LSB is 0, STRB0_B0 is asserted during the write cycle. The remaining bits of the original logical address are placed on the external address bus starting at pin A_{-1} (because the memory width is 16 bits).

Summary: 16/8-Bit Memory Configuration Design Examples

Two conclusions can be drawn from these examples. First, while designing the external memory interface to the TMS320C32, a hardware engineer must remember to match address pin A_{-1} of the ’C32 with the A0 pin of a 16-bit-wide memory or to match the A_{-2} address pin of the ’C32 with the A0 pin of a byte-wide memory. If the external memory is 32 bits wide, the pins are not shifted relative to each other and match perfectly at A0.

Second, when writing code for the ’C32, the programmer does not have to be concerned about the structure of the physical memory. The programmer must simply be aware of the logical memory map and the configuration of the two strobe control registers. Furthermore, all the address translation tasks and byte packing/unpacking necessary to match variable-size data with physical memories of different widths are automatically performed by the ’C32 memory interface and controlled by the data size and memory width fields of the STRB0 and STRB1 control registers.



8-bit Data Size – Address Shifted By 2 bits

Figure 14. 16/8-Bit Memory Address Translation: Data Size < Memory Width

ONE BANK/TWO STROBES (32-BIT-WIDE MEMORY) DESIGN EXAMPLES

The following examples describe how to use two strobes in interfacing the TMS320C32 to a single physical bank of memory. Such configuration enables the access to 32-bit programs and two differently sized portions of data out of the same bank of memory with no speed penalty. This feature is implemented by internally ANDing $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ and outputting the combined strobes on $\overline{\text{STRB0}}$ (a total of four lines). The one bank/two strobes memory configuration is useful in systems where, for example, the program requiring 32-bit instruction words for maximum execution speed operates on data that needs only 16 bits of precision (see Figure 18 on page 35).

Figure 15 is the schematic diagram of a 32-bit-wide external memory configuration arranged as one bank with two separate logical control strobes sharing the same $\overline{\text{STRB0}}$ physical signal lines. The four $\overline{\text{STRB0}}$ signals are assigned to the chip-select pins of four 32K $\times$ 8 15-ns SRAMs, one signal per chip. For the 60-MHz version of the ’C32, the 15-ns SRAMs operate at zero wait states (for slower devices, additional software wait states can be programmed in the appropriate fields of the strobe control registers). Because the total memory width is 32 bits, there is no mismatch between the processor’s and the memory’s address pins. Therefore, the ’C32 pin A0 is matched with memory pin A0; A1 is matched with A1; and so on. As mentioned earlier, both $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ signals appear together on the four $\overline{\text{STRB0}}$ control pins. This is selected by setting the strobe configuration bit of the $\overline{\text{STRB0}}$ control register to 1 (see Figure 15). Since both $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ are mapped to different ranges of the logical memory map, the strobe that actually appears on the physical $\overline{\text{STRB0}}$ pins depends on the internal address of the data/program being accessed. The two strobes effectively split the physical memory in two, with the high memory address bit selecting either the $\overline{\text{STRB0}}$ or $\overline{\text{STRB1}}$ address space. For example, if all program instructions were fetched from logical addresses 880000h – 881000h and all data reads/writes were confined between 980000h and 981000h, the program fetches would be associated with $\overline{\text{STRB0}}$ and all data accesses would be driven by $\overline{\text{STRB1}}$ (see Figure 1 on page 3 for strobe/memory mapping). Since the behavior of each strobe is determined by a different control register, the program fetches and data reads/writes in each case can vary in how many $\overline{\text{STRB0}}$ lines are simultaneously driven and in the number of bus cycles required per access, as illustrated by the following examples.

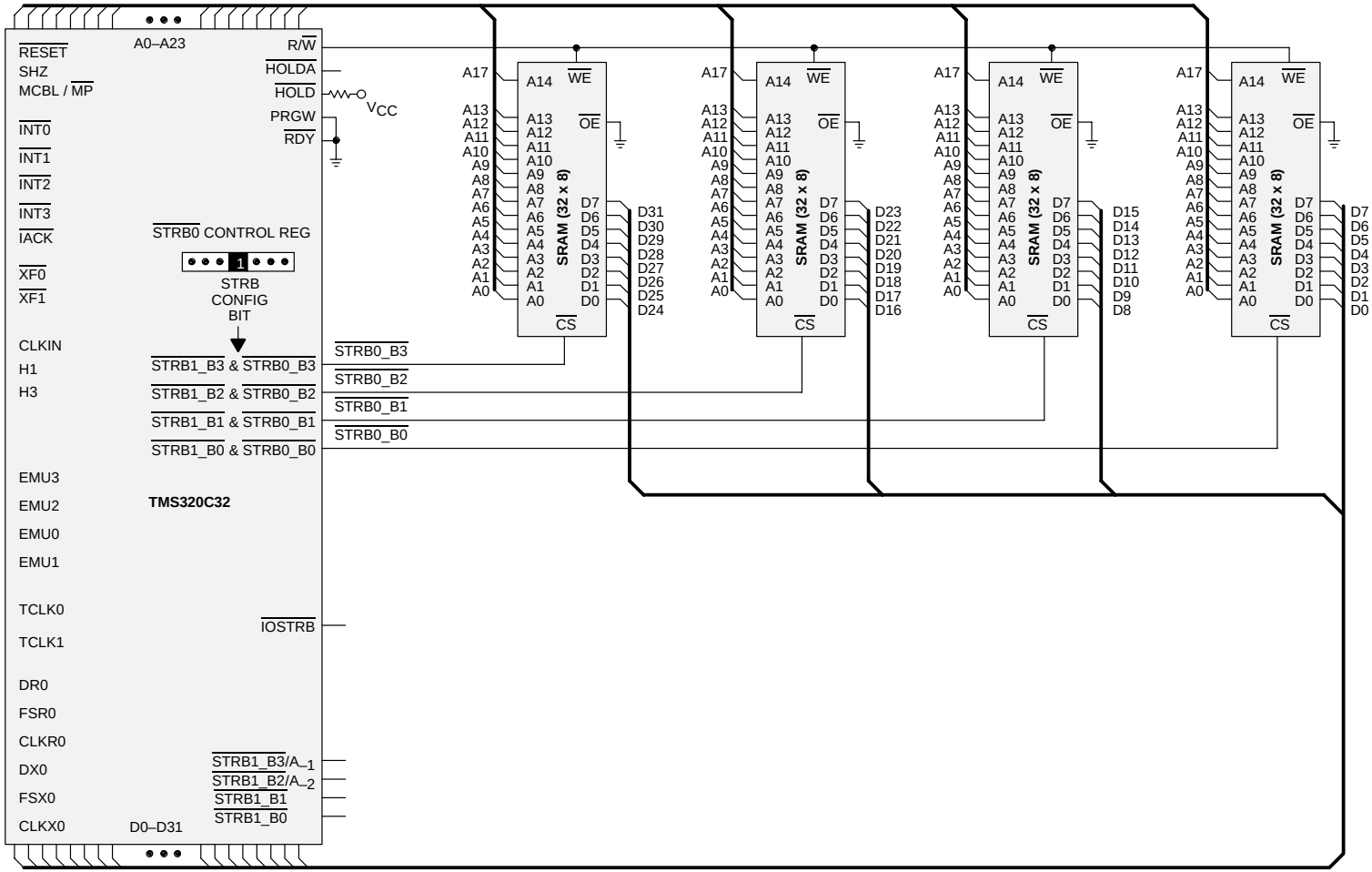


Figure 15. One Bank/Two Strobes Memory Configuration: Memory Width = 32 Bits

Example 10. One Bank/Two Strobes Address Translation for Data Size = 16 and 8 Bits

Figure 16 illustrates how a single physical block of memory can be split into two separate logical halves, one with 16-bit data and the other with 8-bit data. The access to each half is controlled by a separate strobe control register with corresponding memory width and data size fields. Another $\overline{\text{STRB0}}$ control register field, STRB CONFIG (strobe configuration), is set to 1 to indicate that both $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ are mapped to the same set of four $\overline{\text{STRB0}}$ pins. As stated previously, the high memory address pin (in this case, A14) selects between the two halves of the memory. For this example, the ’C32 address pin A17 was chosen to drive the memory pin A14.

The state of the A17 bit of the physical address is derived from the logical address (logical as seen by the instruction). The state of the A17 bit also depends on the logical/physical address shift as determined by the size of the program/data that is being accessed. In this case, the logical $\overline{\text{STRB0}}$ address range is deliberately chosen to drive the physical address bit A17 to 0 (after accounting for a 1-bit address shift due to the 16-bit width of the data). Similarly, the logical $\overline{\text{STRB1}}$ range is chosen to drive the physical address bit A17 to 1 (after accounting for a 2-bit address shift due to the 8-bit width of the data). Additionally, the logical $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ address ranges that were selected to drive the physical address pin A17 to 0 and 1 still have to conform to the logical memory map that assigns fixed blocks of addresses to different strobe spaces.

To the programmer writing software for this memory configuration, this simply means that an STI R0,*AR0 instruction (with AR0 = 887FFFh) results in a $\overline{\text{STRB0}}$ data access (data size = 16 bits) driving the $\overline{\text{STRB0\_B2}}$ and $\overline{\text{STRB0\_B3}}$ control pins to write the contents of the 32-bit register R0 into a 16-bit data location in the lower half of the external memory addressed by 3FFFh. Similarly, an LDI *AR1,R1 instruction (with AR1 = 98FFFFh) results in a $\overline{\text{STRB1}}$ data access (data size = 8 bits) driving the $\overline{\text{STRB0\_B3}}$ (because STRB CONFIG = 1) control pin to read the contents of an 8-bit data location in the upper half of the external memory addressed by 7FFFh to the 32-bit R1 register. Once again, all address translation is performed automatically by the ’C32, and the programmer merely has to watch the logical memory map and the two strobe control registers.

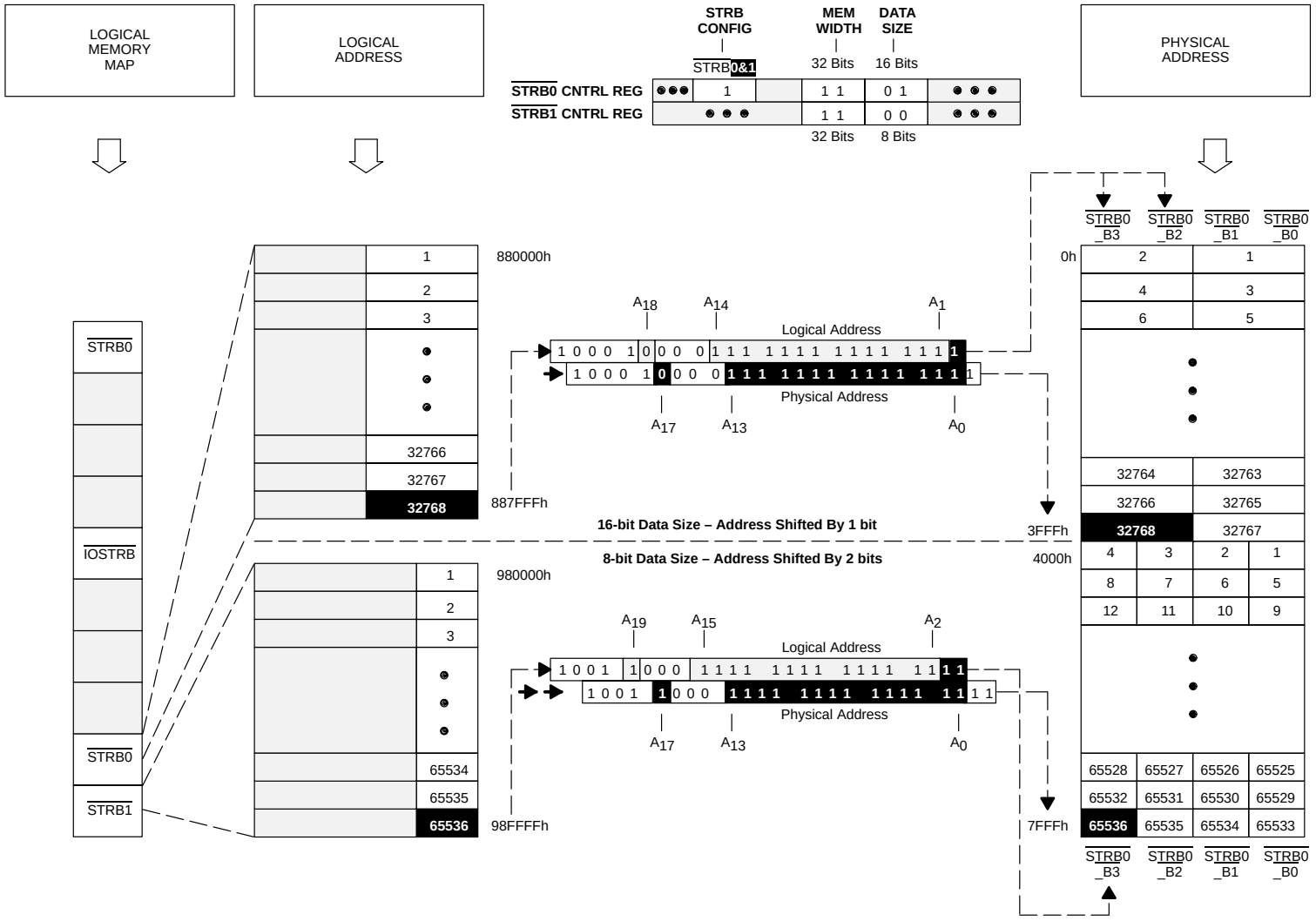


Figure 16. One Bank/Two Strobes Address Translation: Data Size = 16 and 8 Bits

Example 11. One Bank/Two Strobes Address Translation for Data Size = 32 and 8 Bits

Figure 17 illustrates how a single physical block of memory can be split into two separate logical halves, one with 32-bit data and the other with 8-bit data. The access to each half is controlled by a separate strobe control register with corresponding memory width and data size fields. Another $\overline{\text{STRB0}}$ control register field, STRB CONFIG, is set to 1 to indicate that both $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ are mapped to the same set of four $\overline{\text{STRB0}}$ pins. As stated previously, the high memory address pin (in this case, A14) selects between the two halves of the memory. For this example, the 'C32 address pin A17 was chosen to drive the memory pin A14.

The state of the A17 bit of the physical address is derived from the logical address (logical as seen by the instruction). The state of the A17 bit also depends on the logical/physical address shift as determined by the size of the program/data that is being accessed. In this case, the logical $\overline{\text{STRB0}}$ address range is deliberately chosen to drive the physical address bit A17 to 0. Similarly, the logical $\overline{\text{STRB1}}$ range is chosen to drive the physical address bit A17 to 1 (after accounting for a 2-bit address shift due to the 8-bit width of the data). Additionally, the logical $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ address ranges that were selected to drive the physical address pin A17 to 0 and 1 still have to conform to the logical memory map that assigns fixed blocks of addresses to different strobe spaces.

To the programmer writing software for this memory configuration, this simply means that an STI R0,*AR0 instruction (with AR0 = 883FFFh) results in a $\overline{\text{STRB0}}$ data access (data size = 32 bits) driving the $\overline{\text{STRB0\_B0}}$, $\overline{\text{STRB0\_B1}}$, $\overline{\text{STRB0\_B2}}$, and $\overline{\text{STRB0\_B3}}$ control pins to write the contents of the 32-bit register R0 into a 32-bit data location in the lower half of the external memory addressed by 3FFFh. Similarly, an LDI *AR1,R1 instruction (with AR1 = 98FFFFh) results in a $\overline{\text{STRB1}}$ data access (data size = 8 bits) driving the $\overline{\text{STRB0\_B3}}$ (because STRB CONFIG = 1) control pin to read the contents of an 8-bit data location in the upper half of the external memory addressed by 7FFFh to the 32-bit R1 register. Once again, all address translation is performed automatically by the 'C32, and the programmer merely has to watch the logical memory map and the two strobe control registers.

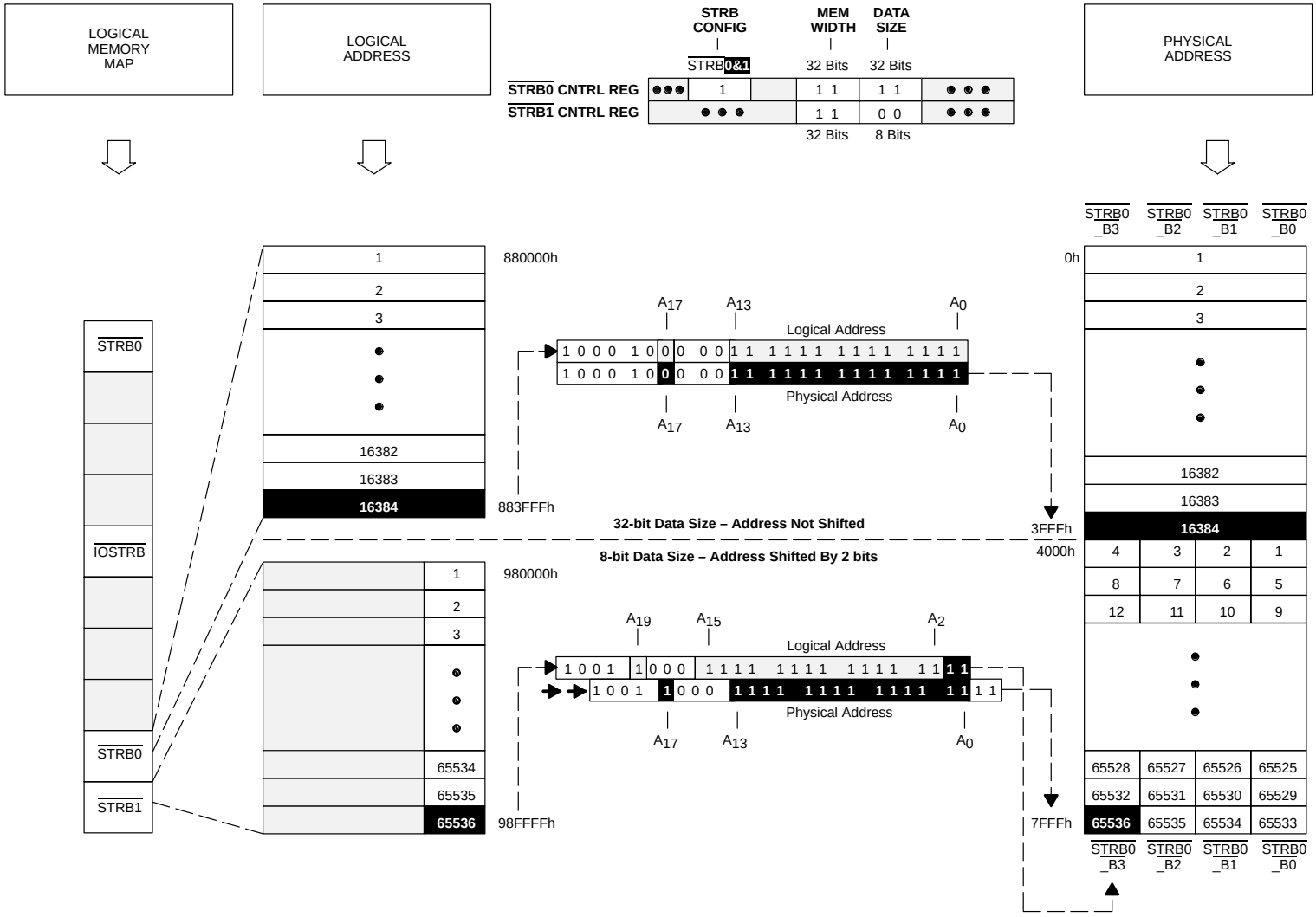


Figure 17. One Bank/Two Strobes Address Translation: Data Size = 32 and 8 Bits

Example 12. One Bank/Two Strobes Address Translation for Data Size = 16 and 32 Bits

Figure 18 illustrates how a single physical block of memory can be split into two separate logical halves, one with 16-bit data and the other with 32-bit data. The access to each half is controlled by a separate strobe control register with corresponding memory width and data size fields. Another $\overline{\text{STRB0}}$ control register field, STRB CONFIG, is set to 1 to indicate that both $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ are mapped to the same set of four $\overline{\text{STRB0}}$ pins. As stated previously, the high memory address pin (in this case, A14) selects between the two halves of the memory. For this example, the ’C32 address pin A17 was chosen to drive the memory pin A14.

The state of the A17 bit of the physical address is derived from the logical address (logical as seen by the instruction). The state of the A17 bit also depends on the logical/physical address shift as determined by the size of the program/data that is being accessed. In this case, the logical $\overline{\text{STRB0}}$ address range is deliberately chosen to drive the physical address bit A17 to 0 (after accounting for a 1-bit address shift due to the 16-bit width of the data). Similarly, the logical $\overline{\text{STRB1}}$ range is chosen to drive the physical address bit A17 to 1. Additionally, the logical $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ address ranges that were selected to drive the physical address pin A17 to 0 and 1 still have to conform to the logical memory map that assigns fixed blocks of addresses to different strobe spaces.

To the programmer writing software for this memory configuration, this simply means that an STI R0,*AR0 instruction (with AR0 = 887FFFh) results in a $\overline{\text{STRB0}}$ data access (data size = 16 bits) driving the $\overline{\text{STRB0\_B2}}$ and $\overline{\text{STRB0\_B3}}$ control pins to write the contents of the 32-bit register R0 into a 16-bit data location in the lower half of the external memory addressed by 3FFFh. Similarly, an LDI *AR1,R1 instruction (with AR1 = 923FFFh) results in a $\overline{\text{STRB1}}$ data access (data size = 32 bits) driving the $\overline{\text{STRB0\_B0}}$, $\overline{\text{STRB0\_B1}}$, $\overline{\text{STRB0\_B2}}$, and $\overline{\text{STRB0\_B3}}$ (because STRB CONFIG = 1) control pins to read the contents of a 32-bit data location in the upper half of the external memory addressed by 7FFFh to the 32-bit R1 register. Once again, all address translation is performed automatically by the ’C32, and the programmer merely has to watch the logical memory map and the two strobe control registers.

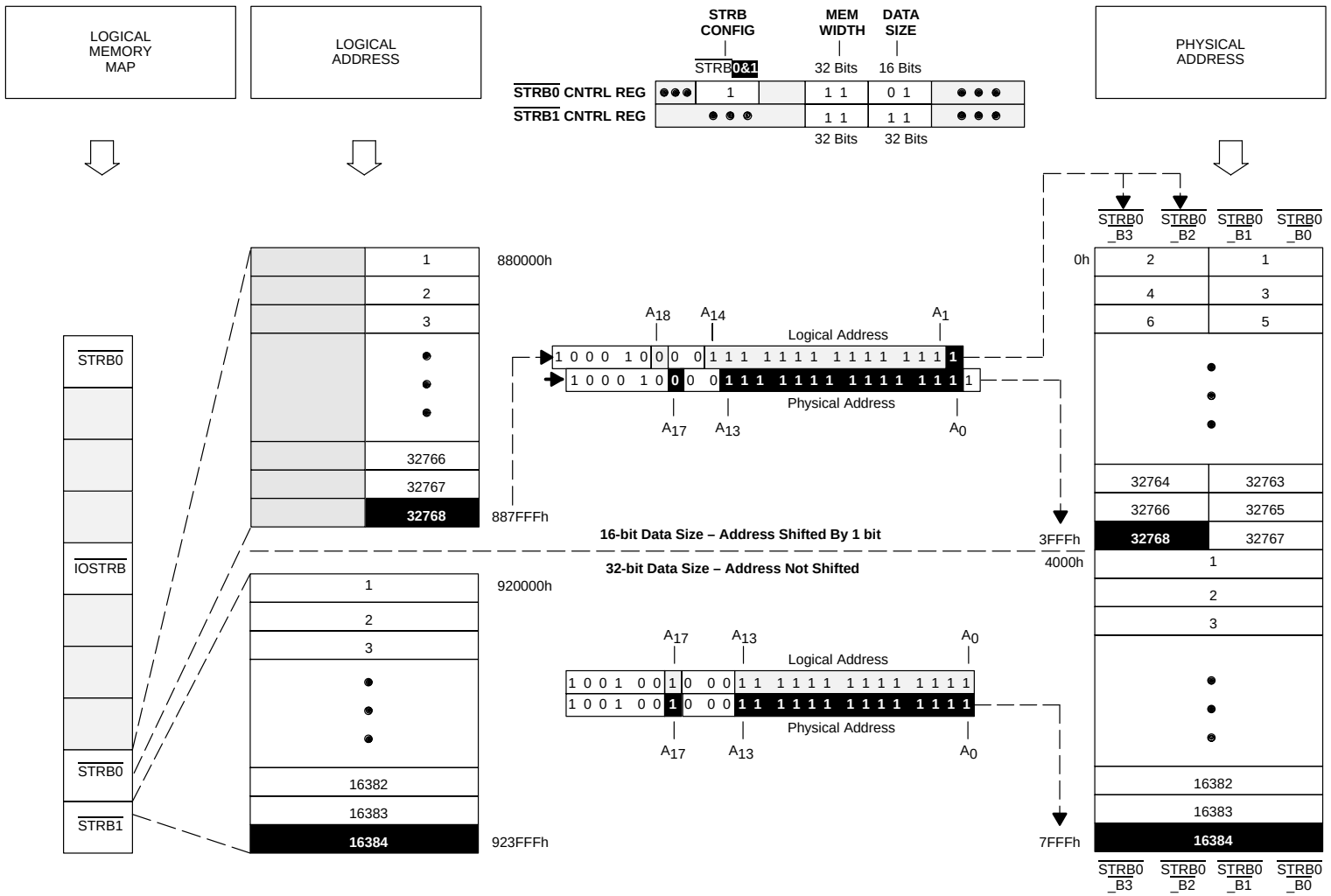


Figure 18. One Bank / Two Strobes Address Translation: Data Size = 16 and 32 Bits

Summary: One Bank/Two Strobes (32-Bit Memory) Design Examples

To summarize these examples, the one bank/two strobes memory interface to the ’C32 supports any combination of data size pairs (16/8, 32/8, and 16/32 bits) with no speed penalty (the strobe control registers do not have to be reconfigured each time data size changes). Likewise, a 16-bit external memory can be divided into two halves, each containing data of a different size (8, 16, or 32 bits). The same holds true for 8-bit external memory. All address translation information given in Examples 1 through 9 applies to the one bank/two strobes examples also.

To configure the external memory for one bank/two strobes access mode, the following steps are recommended :

1. Set the strobe configuration field in the $\overline{\text{STRB0}}$ control register to 1.
2. Set the memory width field in both the $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ control registers to reflect the width of the physical memory.
3. Set the data size field in both the $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ control registers to reflect the size of the data portions chosen for each strobe.
4. Choose one of the high physical address bits to split the physical memory into two halves.
5. For the two memory halves, choose the $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ logical address ranges to drive the chosen bit to 0 and 1, respectively. The chosen $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ address ranges have to fit inside the legal $\overline{\text{STRB0}}$/$\overline{\text{STRB1}}$ address spaces as defined by the memory map.

$\overline{\text{RDY}}$ SIGNAL GENERATION

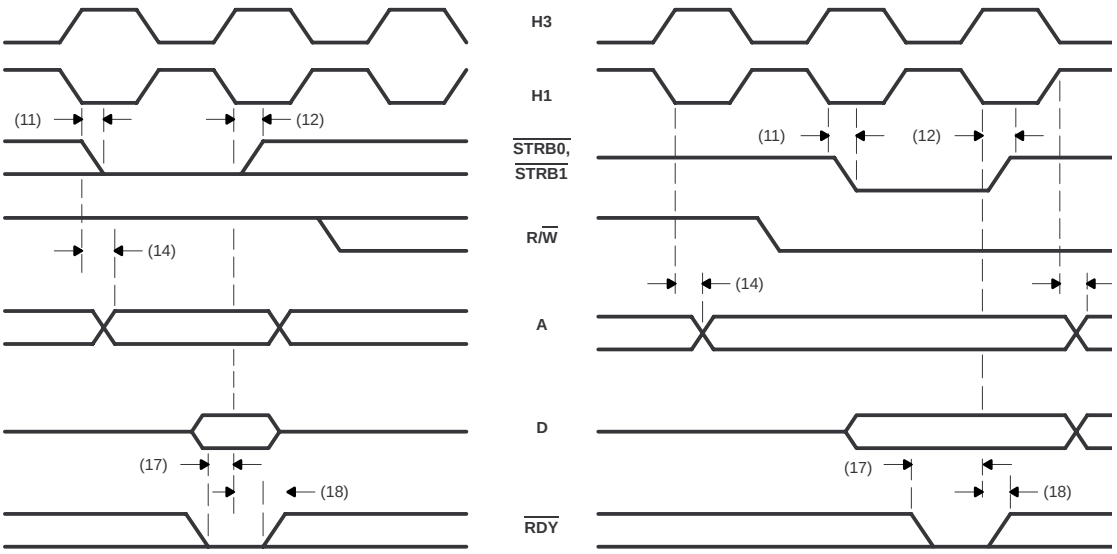
The ’C32 uses the $\overline{\text{RDY}}$ pin to determine whether the current bus cycle will finish at the end of the current clock cycle or require additional clock cycles to complete. Even though the ’C32 is capable of fetching instructions and accessing data in one clock cycle, a slow memory may need additional clock cycles (wait states) to complete the bus cycle. The generation of the $\overline{\text{RDY}}$ signal can be accomplished in three ways:

1. In many systems, all external memory is fast enough to preclude wait states. In these cases, the $\overline{\text{RDY}}$ pin can be permanently grounded, indicating to the CPU that the external memory is always ready for the next cycle.
2. Even if there are external devices that require wait states, as long as there is only one device per strobe, the wait states can be programmed in software by setting bits in corresponding strobe control registers. As in the first case, the $\overline{\text{RDY}}$ pin should be permanently grounded.
3. The active generation of the $\overline{\text{RDY}}$ signal is required only if a single strobe controls two or more external memory banks or peripherals requiring different numbers of wait states.

The remainder of this section addresses the third method. The example involves three memory banks controlled by $\overline{\text{STRB0}}$, each requiring a different number of wait states. Note that this example directly applies to $\overline{\text{RDY}}$ signal generation involving $\overline{\text{STRB1}}$ and is similar to the case of $\overline{\text{IOSTRB}}$, which involves a more relaxed set of timing parameters.

$\overline{\text{RDY}}$ Signal Timing Parameters for $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$

Figure 19 contains $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ timing parameters that would typically be used to generate the $\overline{\text{RDY}}$ signal. As evident in the read and write timing waveforms, the $\overline{\text{RDY}}$ signal generated by the external logic is clocked into the ’C32 on the falling edge of the H1 clock. The associated setup time is represented by parameter (17) and the hold time by parameter (18). So, for the 60-MHz ’C32, the $\overline{\text{RDY}}$ signal must arrive at the $\overline{\text{RDY}}$ pin at least 17 ns before the falling edge of H1 and remain valid at least until H1 goes low. Timing parameters (11) and (12) represent the $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ low and high delays from the falling edge of H1. Timing parameter (14) represents the address valid delay from the falling edge of H1. For back-to-back write cycles, timing parameter (22) represents the address valid delay from the rising edge of H1. Parameters (11), (12), (14), and (22) do not directly apply to $\overline{\text{RDY}}$ setup and hold, but are nevertheless involved in the generation of the $\overline{\text{RDY}}$ signal.



NO.	DESCRIPTION		'320C32-40† (50 ns)		'320C32-50† (40 ns)		'320C32-60† (33 ns)		UNIT
			MIN	MAX	MIN	MAX	MIN	MAX	
11	$t_{d(H1L-SL)}$	Delay time, H1 low to $\overline{\text{STRBx}}$ low	0	11	0	9	0	8	ns
12	$t_{d(H1L-SH)}$	Delay time, H1 low to $\overline{\text{SRBx}}$ high	0	11	0	9	0	8	ns
14	$t_{d(H1L-A)}$	Delay time, H1 low to A valid	0	11	0	9	0	8	ns
17	$t_{su(RDY)}$	Setup time, $\overline{\text{RDY}}$ before H1 low	21		19		17		ns
18	$t_{h(RDY)}$	Hold time, $\overline{\text{RDY}}$ after H1 low	0		0		0		ns
22	$t_{d(H1H-A)}$	Delay time, H1 high to A valid on back-to-back write cycles (write)		11		9		8	ns

† These timing specifications are subject to change without notice. See the TMS320C32 Data Sheet for current timing information.

Figure 19. $\overline{\text{RDY}}$ Signal Timing for $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ Cycles

Example 13. $\overline{\text{RDY}}$ Signal Generation

The example in Figure 20 involves three memory banks controlled by a single strobe ($\overline{\text{STRB0}}$). The first bank is composed of four 8-bit-wide SRAMs requiring zero wait states to operate at 60 MHz (15-ns devices). Bank 2 is composed of two 1-wait-state SRAMs, and bank 3 contains one 3-wait-state EPROM (which is eight bits wide). The $\overline{\text{RDY}}$ pin is normally high, indicating a not-ready state. It goes low if either $\overline{\text{RDY\_BANK1}}$ or $\overline{\text{RDY\_BANK23}}$ goes low.

The $\overline{\text{RDY\_BANK1}}$ signal is asserted only if two conditions are satisfied. First, at least one of the four $\overline{\text{STRB0}}$ signal lines must be active. Second, the three address decode bits must match the bank 1 space. Since no wait states are involved, the $\overline{\text{RDY\_BANK1}}$ signal does not have to be synchronized with the H1/H3 clocks, and, therefore, it can directly drive the $\overline{\text{RDY}}$ pin after being gated with its bank 2/bank 3 counterpart.

The STRB0_BANK23 signal becomes active (high) if the three address decode bits match bank 2 or bank 3 address spaces while $\overline{\text{STRB0\_B0}}$ and/or $\overline{\text{STRB0\_B1}}$ are active (low). The STRB0_BANK23 signal, when high, sets a high data state in a synchronous progression through a chain of four registers. Depending on which point in the chain is tapped, a $\overline{\text{RDY}}$ signal delay ranging from zero to three wait states can be achieved. In this case, both 1-wait-state and 3-wait-state taps assert the $\overline{\text{RDY\_B23YES}}$ signal to reflect bank 2 or bank 3 access. Finally, a two-register circuit shaves the trailing edge of the $\overline{\text{RDY\_B23YES}}$ signal by ORing it with RDY_23NOT (see Figure 21). The resulting $\overline{\text{RDY\_BANK23}}$ is ANDed with its bank 1 counterpart to drive the $\overline{\text{RDY}}$ pin.

Figure 21 contains timing waveforms for the $\overline{\text{RDY}}$ generation example. It illustrates how the $\overline{\text{RDY}}$ signal is generated for a series of external back-to-back memory read cycles in which the first one accesses bank 1 (zero wait states), the second accesses bank 2 (one wait state), the third accesses bank 3 (three wait states), and the fourth and fifth access bank 1 (zero wait states). For each read cycle, the $\overline{\text{RDY}}$ waveform is marked with a resulting setup time. For the 60-MHz device, the $\overline{\text{RDY}}$ signal should become valid at least 17 ns before every falling edge of the H1 clock.

In the 0-wait-state cycle, the address and strobe signals become valid 8 ns from the falling edge of H1 (see Figure 21). An additional 5 ns are needed for a single pass through a fast combinational logic device for a total setup time of the resulting $\overline{\text{RDY}}$ signal equal to 20 ns, leaving 3 ns for board delays and a modest safety factor.

For the 1- and 3-wait-state cycles, the bank decode and strobe signals do not directly drive the $\overline{\text{RDY}}$ signal. They are instead combined into the STRB0_BANK23 signal that, when active, releases the clear condition on the 3-register delay chain driven by the H3 clock. The register chain is then free to propagate a high state at the rate of one register per clock cycle. The two taps in the register chain (at the first and third registers, representing one wait and three wait states, respectively) are ORed with their corresponding bank select signals to result in the $\overline{\text{RDY\_B23YES}}$ signal synchronous to H1/H3 clocks. The $\overline{\text{RDY\_B23YES}}$ leading-edge 10-ns delay is caused by two passes through a fast PAL® device (such as a popular 22V10). The trailing edge of this signal is caused by bank 2 or bank 3 decode circuits going inactive after the $\overline{\text{RDY}}$ signal is recognized by the processor. The address decode (8 ns) plus two passes through the PAL (5 + 5 ns) combine for a total delay of 18 ns that, if not modified, would cut into the next cycle's $\overline{\text{RDY}}$ setup requirement (33 – 18 = 15 ns). To deactivate the $\overline{\text{RDY}}$ signal sooner, a single-register circuit has been added to generate the RDY_B23NOT, which, when ORed with the $\overline{\text{RDY\_B23YES}}$, yields the $\overline{\text{RDY\_BANK23}}$ signal that satisfies the $\overline{\text{RDY}}$ setup time for the next cycle. Finally, $\overline{\text{RDY\_BANK1}}$ and $\overline{\text{RDY\_BANK23}}$ are ANDed together to produce the final $\overline{\text{RDY}}$ signal that is wired to the processor's $\overline{\text{RDY}}$ pin.

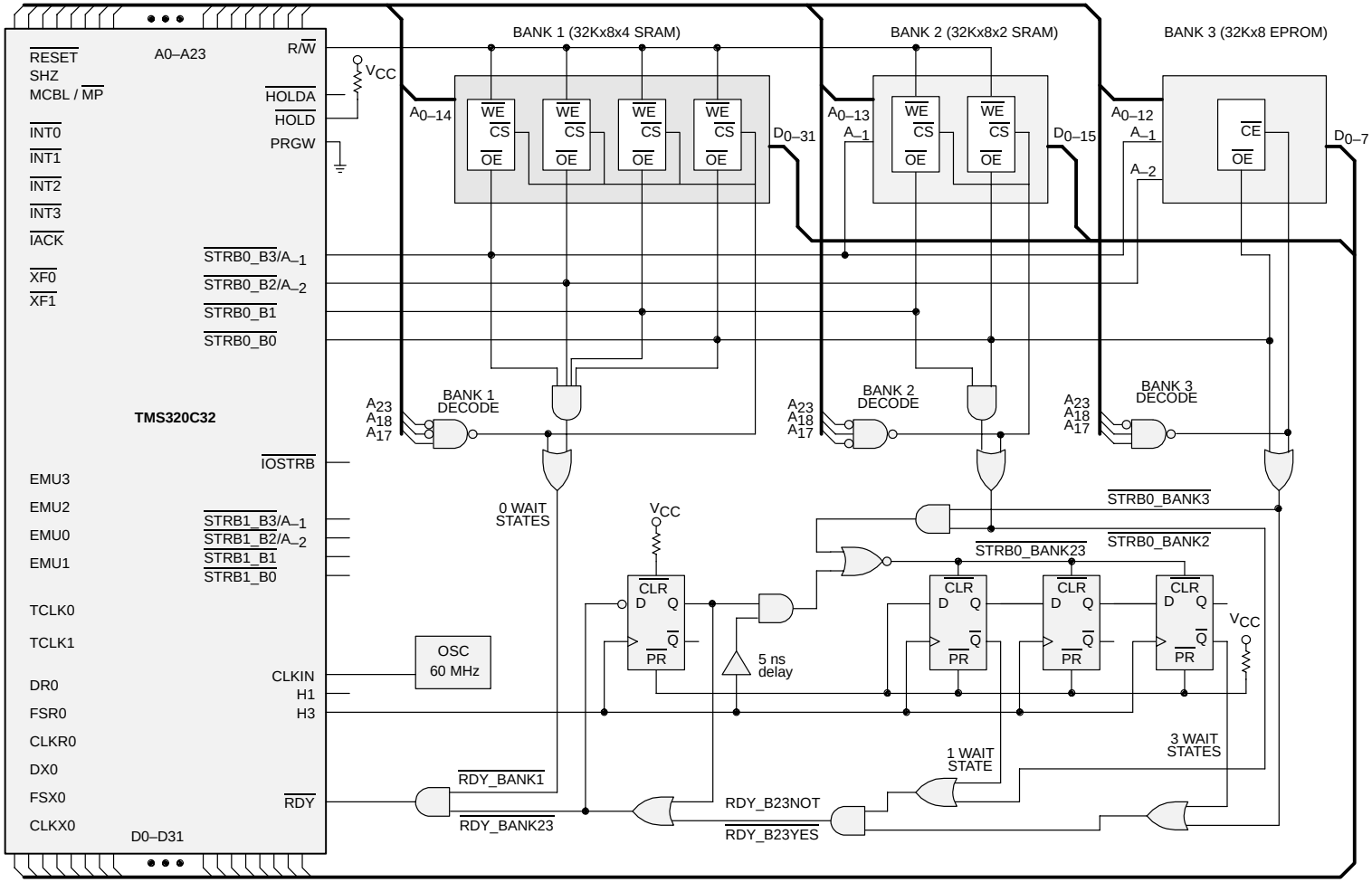


Figure 20. $\overline{\text{RDY}}$ Signal Generation for $\overline{\text{STRB0}}$ Cycles

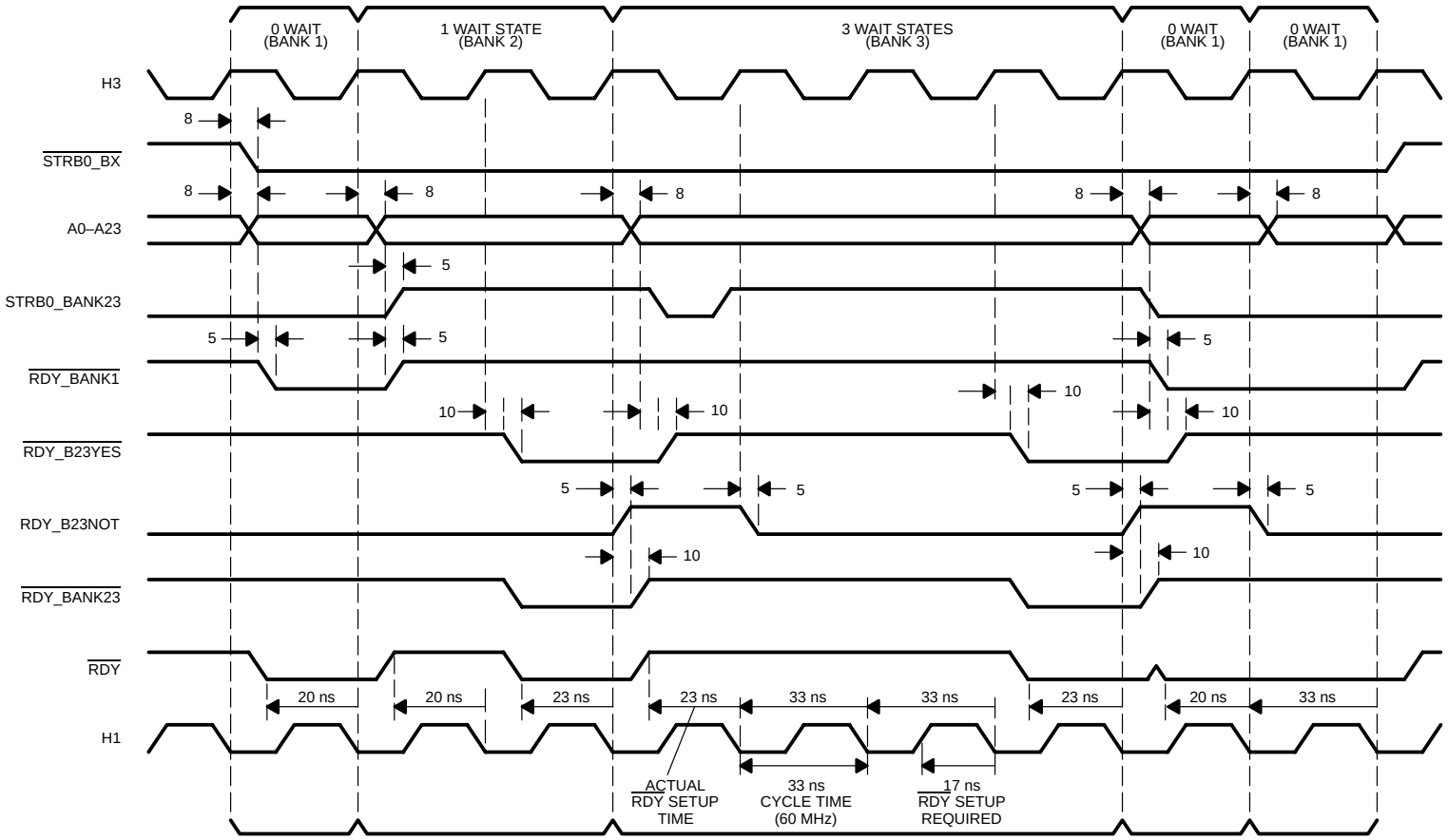


Figure 21. $\overline{\text{RDY}}$ Signal Generation Timing Waveforms

Address Decode for Multiple Banks

Figure 22 illustrates the logical-to-physical address translation for the three memory banks used in the $\overline{\text{RDY}}$ generation example. Each memory bank is of a different physical width, as shown by the external address column on the right side of the figure. The left side of the same figure represents the internal address ranges for each of the three memory banks. Logical-to-physical address translation is controlled by strobe control registers and their data size and memory width fields. The middle column of Figure 22 contains the logical address field (top row) superimposed on the physical address (bottom row) for each address translation case. The active address fields are shaded gray, and the inactive address bits (don't cares) are white. The black fields are special address bits that can selectively control multiple strobe lines or choose between individual portions of a data word that is larger than the physical memory it is accessing.

For example, in bank 2, the right side of the picture indicates that the physical memory width for this bank is 16 bits. The left side indicates that, regardless of the physical memory width, 32-, 16-, and 8-bit data can be moved by programming the $\overline{\text{STRB0}}$ control register. The low-order (shaded) bits of logical/physical address rows show how many bits are actually used for addresses so that the correct high-order address bits can be assigned to bank decode. Physical address bits A17 and A18 were chosen for bank decode because they lie outside the used address bits. A17 and A18 decode between banks 1, 2, and 3, with A18–A17 = (0,1) assigned to bank 1, (1,0) assigned to bank 2, and (1,1) assigned to bank 3. Finally, address bit A23 is set to 0 to isolate the $\overline{\text{STRB0}}$ address space from the $\overline{\text{STRB1}}$ and $\overline{\text{IOSTRB}}$ memory maps.

From the dotted lines bounding the bank decode bits, it is apparent that the external address bits A18–A17 line up perfectly but their logical address counterparts do not. The amount of reverse shift between the logical and physical addresses depends on the size of the data being accessed and the width of the physical memory. Each of the three address translation cases for each of the three banks translates physical address bits A18–A17 into two contiguous logical address bits that can lie anywhere between A20 and A17. Once the logical images of the external bank decode bits have been identified along with low-order address bits and the A23 strobe decode bit, they will together define the final logical memory map for the three $\overline{\text{STRB0}}$ banks.

Once again, each memory bank actually has not one, but three logical memory maps, depending on the size of the data being accessed and the setting of the corresponding bits in the $\overline{\text{STRB0}}$ control register.

The address ranges in these logical memory maps are all different, yet all three maps translate perfectly into a single physical address map that identifies the bank. In using the three logical memory maps, the programmer must exercise caution to prevent overwriting 8-bit data with 16-bit data (or 16-bit data with 32-bit data) that may have a different logical address but still occupy the same place in physical memory. To be certain that the logical address maps associated with 8-, 16-, and 32-bit data sizes do not overlap within a single physical memory bank, the three logical maps should be further divided into mutually exclusive areas before they are used by the programmer. Furthermore, when a program jumps from one physical memory bank to another of a different width, the memory width configuration bits in the appropriate strobe register must be changed.

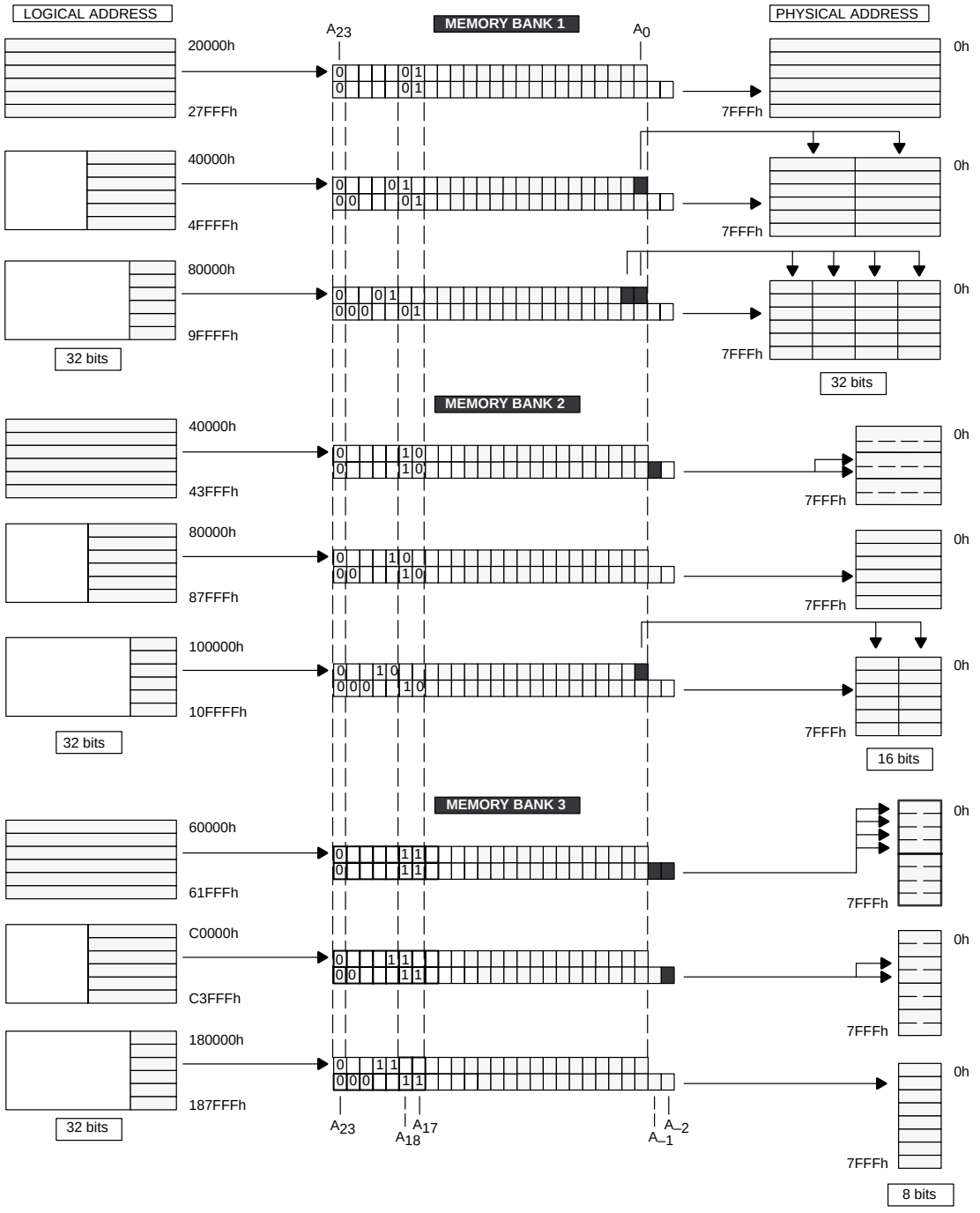


Figure 22. Address Decode for Multiple Memory Banks