Introduction to Designing a System with the DECchip ™ 21064 Microprocessor

Revision/Update Information: Revision 1.0

Digital Equipment Corporation
Maynard, Massachusetts

April, 1992

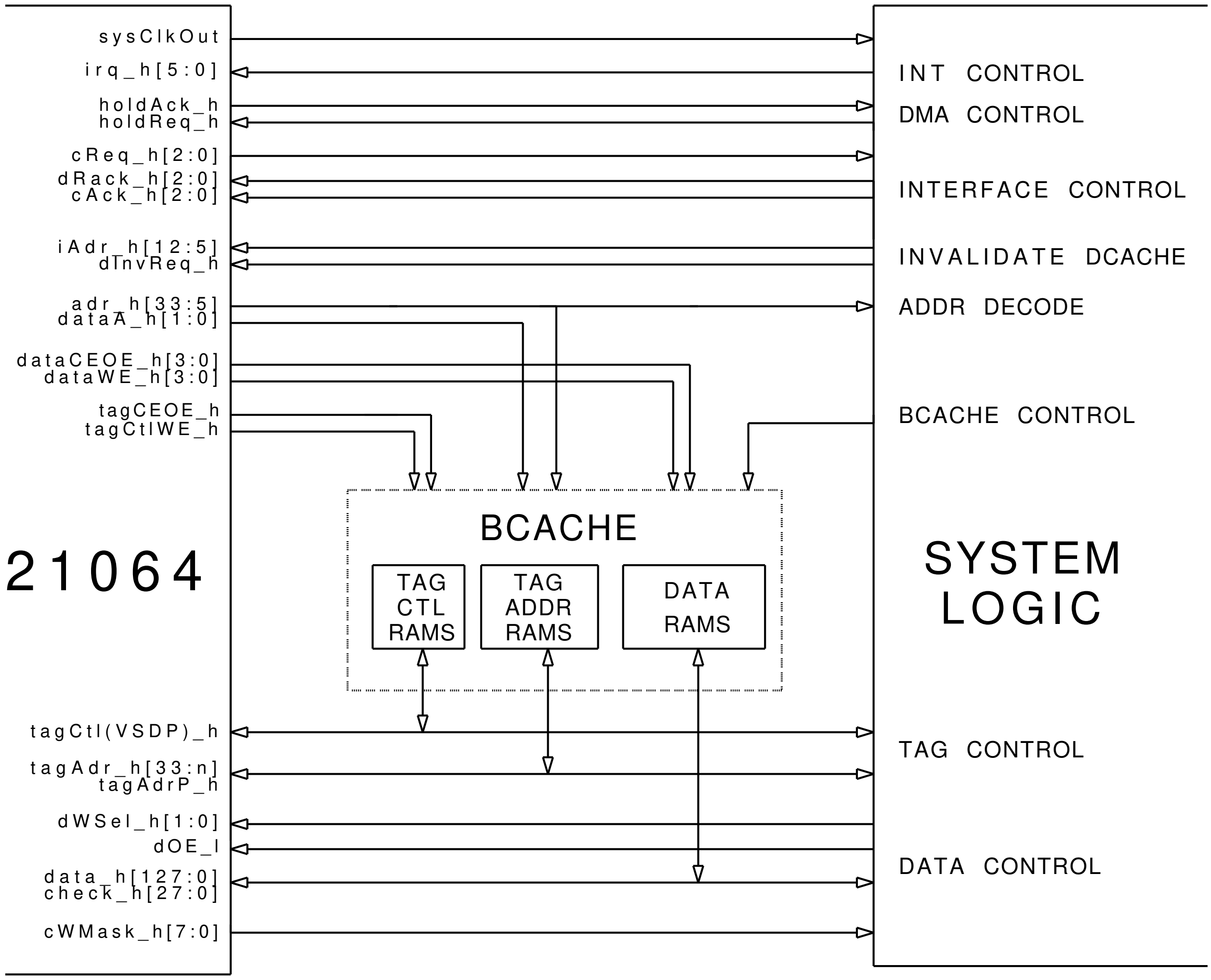
The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

This document was prepared using VAX DOCUMENT, Version 2.0.

1 Introduction

This application note provides a basic description of how to integrate the DECchip ™ 21064 microprocessor chip into a module or system. It describes how the processor reacts to the chip reset condition, and explains how to connect and control the chip interface signals. The 21064 chip has been designed to allow maximum flexibility while at the same time providing the ability to easily create a computing system with generally available module parts.

Figure 1: 21064 Pin Bus



This document is not meant to give every detail about interfacing to the chip. Rather, what is provided here is enough information for a design engineer to understand what is involved in creating a 21064-based system. It should allow the designer to quickly determine how 21064 system design compares with other design tasks.

Examples are used throughout the text to clarify meaning, but this is not intended to imply that what is described is the only way to use the chip. An attempt has been to describe real, usable circuits and techniques, but the chip is flexible and the designer is encouraged to investigate other implementations. A preliminary data sheet is available for the 21064 microprocessor that describes the details and additional features of the chip.

2 General concepts

Some important design concepts are common to many 21064-based system designs, and they are discussed in this section. The chip pin bus is flexible and mandates few design rules, leaving open a wide range of prospective systems. Figure 2 is a diagram of the 21064 pin bus, showing the major signal groups.

A system designed with the chip can be divided into three major sections. There is the 21064 processor itself, the system control logic, and the external backup cache (Bcache) between them (the Bcache is optional, though most systems will see a performance improvement if it is included). The chip pin bus provides address and control signals, and transfers data through a 128-bit bidirectional data bus. The preliminary data sheet describes each of the signals Figure 1 in detail.

The processor controls the Bcache when its initial tag probe finds that the information is valid and unshared. The Bcache access is under control of the CPU, and the external system logic is not involved. When the CPU does a Bcache probe and misses, or when a lock-associated command is invoked, the processor starts an external cycle.

During the external cycle, the Bcache is under control of the system logic. The system logic either returns the data to the processor, or accepts the data from the processor (depending upon the cycle type), and acknowledges the cycle to give control back to the CPU. If the cycle necessitates a Bcache fill, it is up to the system logic to load the data into the Bcache RAMs, the upper address bits (with good parity) into the tag address RAMs, and the proper valid and parity bits into the tag control RAMs.

To help the design engineer create high performance systems more easily, the Bcache is controlled by the 21064 pin bus during probes that hit. This allows off-the-shelf SRAMs to be connected to the chip without a lot of extra components. The Bcache interface signals are programmable through an internal processor register (IPR), so that the Bcache size, access, and write timing can be set with complete flexibility without affecting the internal CPU clock speed.

That is, the 21064 can be running at its nominal 6.6ns internal cycle time, but the Bcache can run slower if required without slowing down the internal timing. There are two internal caches in the 21064 chip: an I-stream read-only cache (Icache) and a D-stream write-through cache (Dcache). The speed of the Bcache does not affect the internal caches, which use the internal clock.

Figure 2 is a block of a system that can be created using the 21064 microprocessor. The major sections are shown, along with many of the buses that would run through such a system. In the center of the diagram is the external interface control, which directs the other system logic subsections that interface to memory, I/O, Bcache, etc.

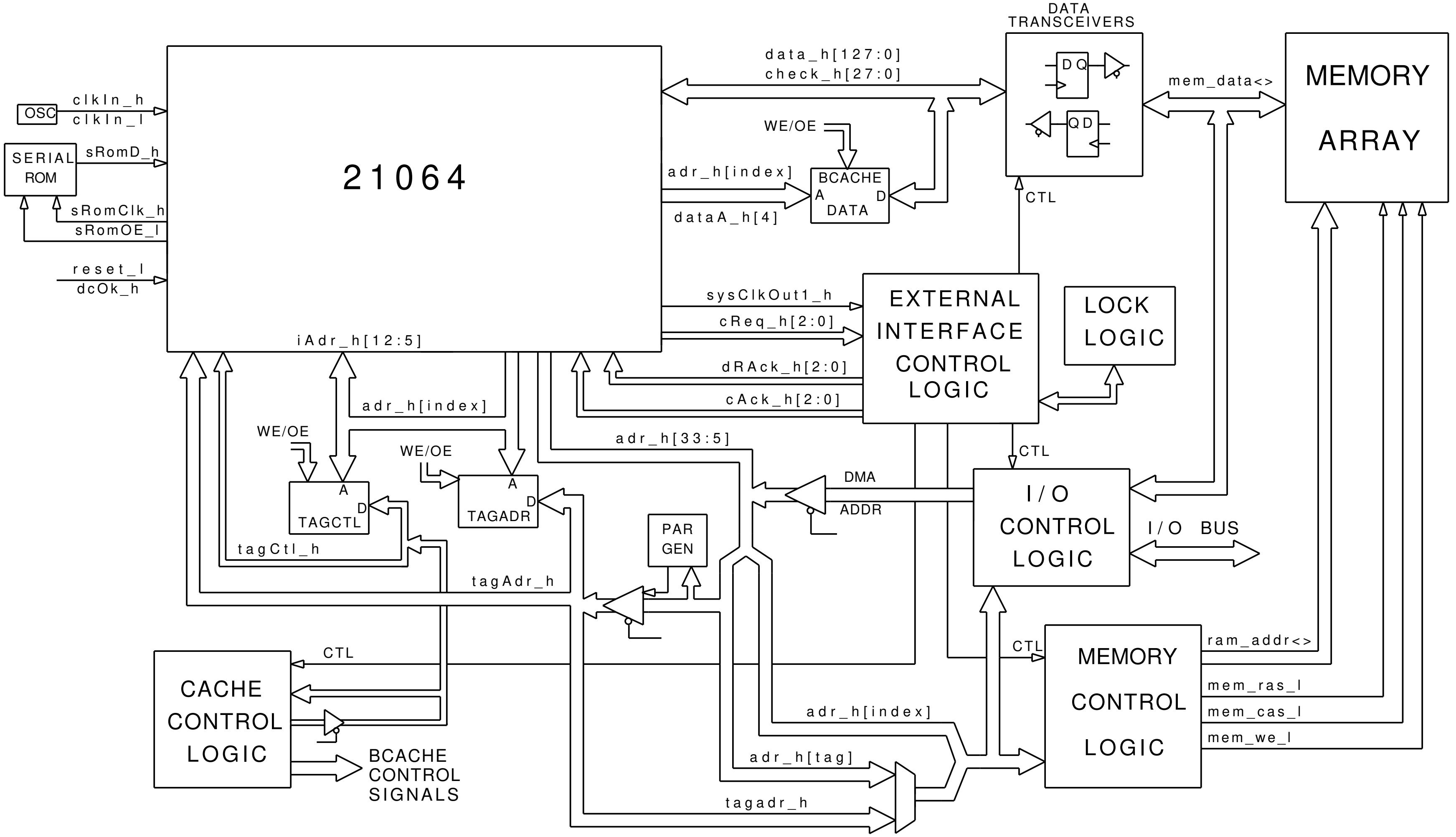
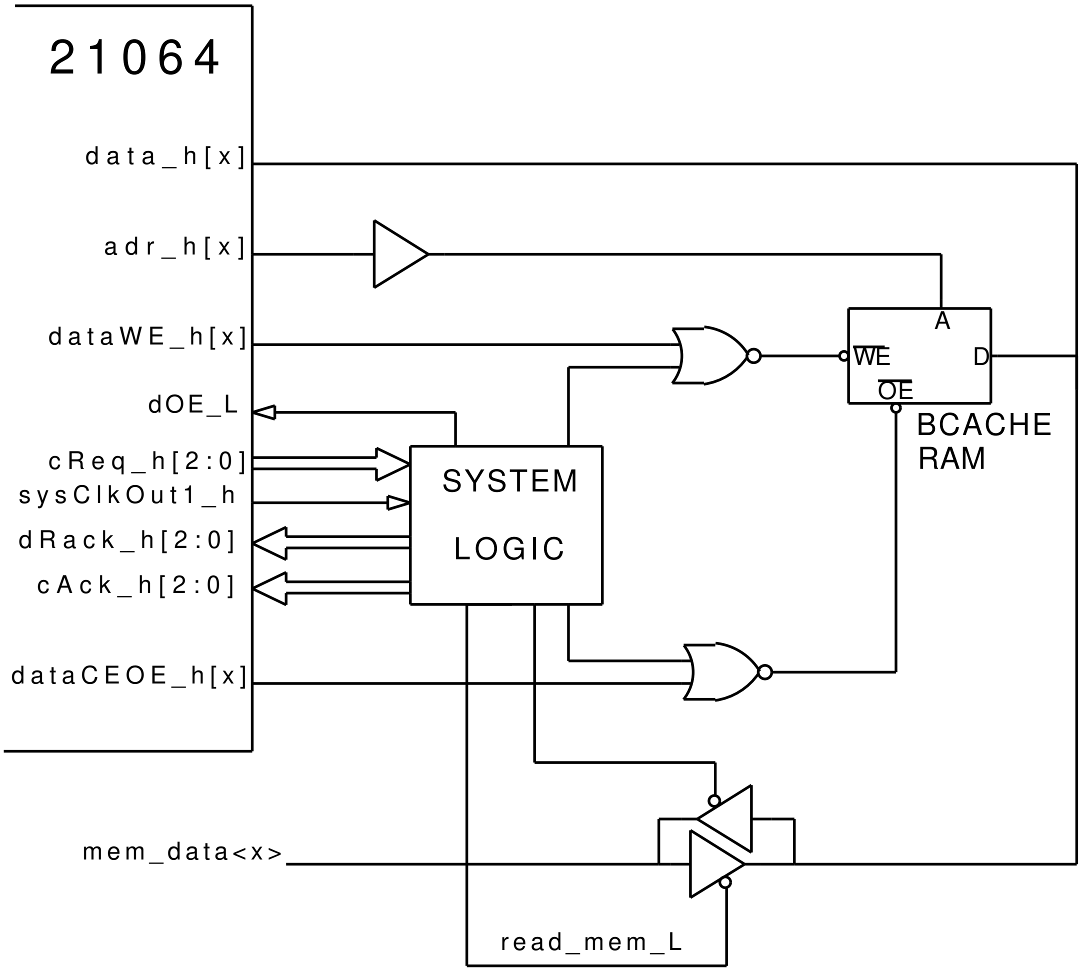


Figure 2: 21064-Based System Block Diagram

The Bcache, if it is included in a system, can be as small as 128KB or as large as 8MB. The size is under program control. The **adr_h[33:5]** bus in Figure 2 is shown partitioned into an [index] field and a [tag] field. The size of each field depends upon the Bcache size. The smallest Bcache (128KB) uses **adr_h[16:5]** to index into the the cache block, and the tag field would be **adr_h[33:17]**. Only those bits that are actually needed for the amount of potentially cachable system main memory need to be stored in the Bcache tag, although the 21064 uses all the relevant tag address bits for that Bcache size on its tag compare. A larger Bcache uses more index bits and fewer tag address bits.

Figure 3: Bcache Control Logic



On an external request (read or write), the 21064 sends out the address and cycle type (and data for a write cycle), then waits until the system logic sends back the acknowledgment handshake that the cycle is complete. On a read request cycle, each data word is tagged as it comes back by the system logic with information about whether the data should be checked for ECC (or parity, depending upon which mode of operation has been selected for the chip), and whether it should be cached inside the chip. On a write request, the system logic merely notifies the chip that the write has been accepted for processing.

The Bcache is shared between the 21064 and the system logic. Although the processor directly manipulates the Bcache for read and write hits, it is up to the system logic to:

- Fill the Bcache with memory data
- Load the tag address and tag address parity
- Load tag control bits and parity on fills (valid and non-dirty)
- Write data back to memory when necessary
- Probe the Bcache for lock/unlock transactions
- Probe and control the Bcache for DMA transactions

The Bcache control signals are thus under potential control of the 21064 or the system logic. When the CPU chip determines that an external cycle is necessary, it drives the Bcache control signals to false. This allows the system logic to read and write the Bcache RAMs. Figure 3 shows the expected configuration for the Bcache. The figure shows a data line, but the tag address and control lines are expected to be connected similarly.

The signal **mem_data** in Figure 3 is a bidirectional memory data bus that connects to the main storage. When it is necessary to load the contents of memory into the Bcache, the system logic will drive the memory bus control signals such that a read cycle is performed. In this example, the signal **read_mem_L** is being used to drive the Bcache (and 21064) data bus. The system logic will properly drive the Bcache RAM write enable signal, and once the data is stable on the **data_h[x]** bus, it will be strobed into the Bcache.

When the Bcache contents need to be written back to memory, the system logic will control the RAM output enable signal to access the Bcache data. The signal **read_mem_L** will now be de-asserted, and the memory control signals will also properly tristate the **mem_data** bus so that the data can be written to the memory storage elements. The system logic must properly assert the 21064 signal **dOE_l** so that the CPU will drive the **data_h[x]** lines.

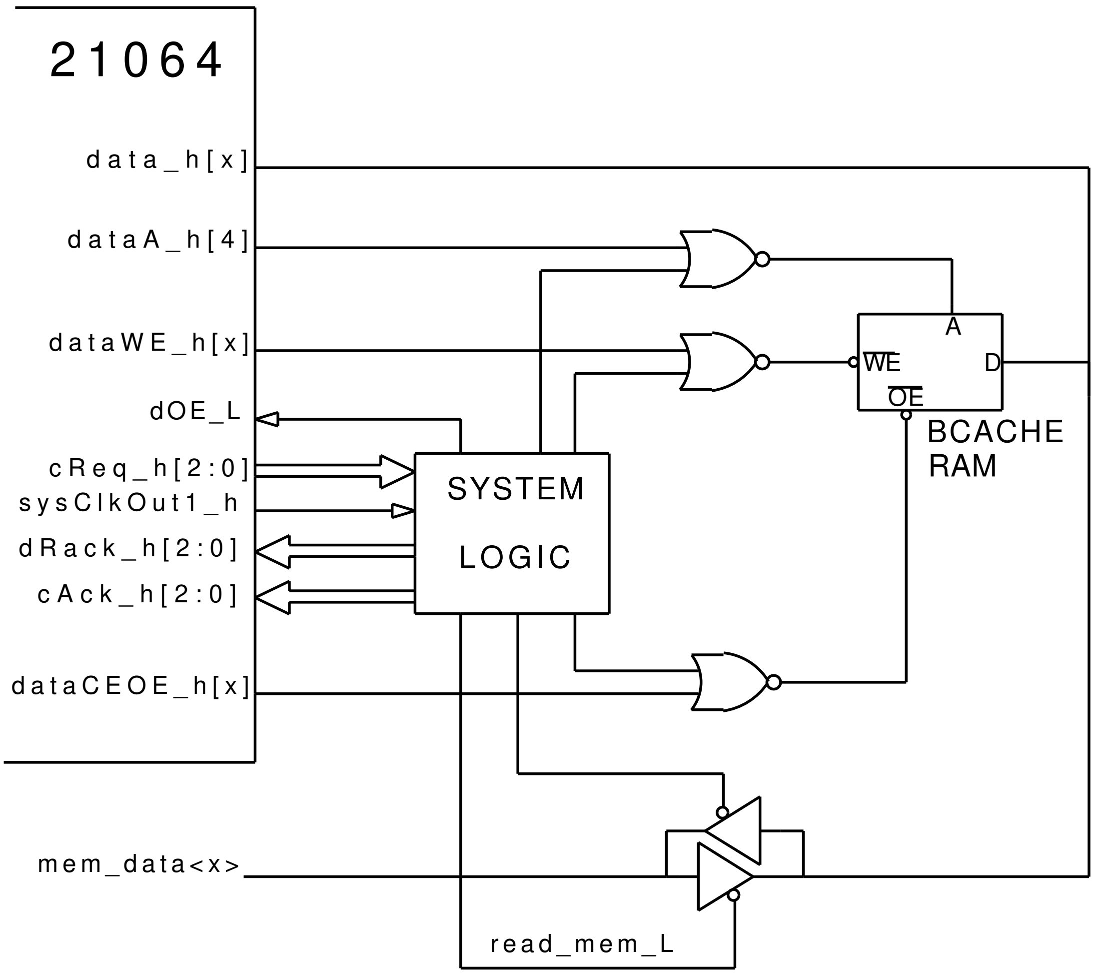
The Bcache consists of 32-byte blocks or larger. As such, the 21064 supplies address bits [33:5] to select which Bcache block. The CPU data bus is 16 bytes wide, and thus each Bcache cycle requires two accesses. The CPU outputs the signal **dataA_h[4]** to control which 16-byte data half is being written to or read from. Figure 4 shows the expected configuration for the lower address bit. As with the chip output enable and write pulse, the lower Bcache address bit is under control of either the 21064 or the system logic. When the CPU is in external system logic mode, it drives the **dataA_h[4]** signal low (along with the other Bcache control signals).

This application note will later go though some general cycle types, including timing diagrams to better explain how a 21064-based system functions.

3 Basic 21064 Power, Input Level, and Clock Issues

The preliminary data sheet describes how to power and clock the 21064 in detail. This section provides an overview of these issues, and some example circuits that can be used.

Figure 4: Lower Bcache Address



3.1 Power Supply and Input Levels

The 21064 is powered from a +3.3V supply (+/- 5%), but will drive and accept CMOS/TTL-compatible levels once the chip has been properly stabilized. It is *mandatory* that no input or bidirectional pin be allowed to rise above 4.0V until the 3.3V power to the chip is stable. Failure to follow this rule will damage the chip.

This rule does *not* imply that power supply sequencing must be used. It only means that any other module part that can drive the input pins must be kept in tristate mode until the 21064 has stable power. So, for example, a *dcOK* signal can be used to prevent components such as SRAMs, MUXes, and buffers from driving the chip.

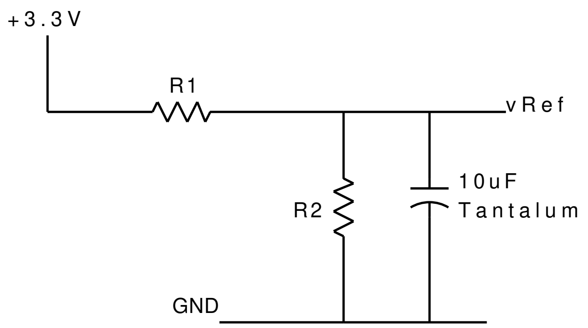
3.2 Input Level Sensing

The 21064 uses a reference input pin **vRef** to supply the threshold level for all chip inputs except:

- clkIn_h,_l
- testclkIn_h,_l
- tagOk_h,_l
- dcOk_h
- eclOut_h
- tristate_l
- cont_l

These pins should never be driven above the 21064 power supply. Since the nominal voltage to the chip is 3.3V, care must be taken if any of the signals above are generated from logic that has a 5V supply. Note especially that **dcOk_h** is one of the signals that must never be driven above the nominal 3.3V level, since it is likely that it will be generated from a higher voltage.

Figure 5: Input Reference Voltage Circuit



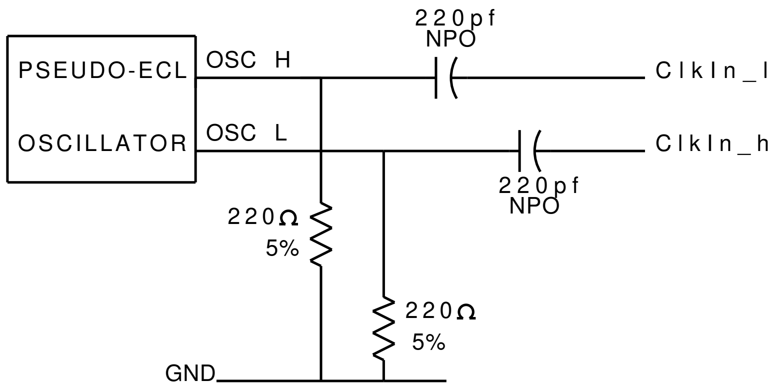
vRef should be connected to a stable 1.4V (+/-10%) source. Figure 5 can be used to supply this voltage level. The resistors R1 and R2 in the figure should be chosen so that they form a voltage divider that supplies the 1.4V level to **vRef**. **vRef** has a large capacitance on it inside the chip, and there is an RC delay between its pin and the other input buffers. Therefore, **dcOk_h** should not be asserted until there has been enough time for the **vRef** input to stabilize. Consult the preliminary data sheet for more details concerning the assertion of **dcOk_h**.

Note that **reset_l** *is* one of the input pins that uses **vRef** for its threshold level, so it cannot be relied upon until **vRef** is stable. **dcOk_h** being false (that is, low) keeps the chip in reset mode.

3.3 Input Clocks

The 21064 expects differential clock signals between 0.6V and 3.0V for the **clkIn_h,_l** inputs. A reversed can pseudo-ECL oscillator with pulldowns can be AC-coupled to the clock inputs for this purpose. Using a pseudo-ECL oscillator means you don't have to design a special ECL power supply to clock the chip. Figure 6 is an example of a working circuit. Note that the series capacitor should use an NPO dielectric.

Figure 6: Input Clock Circuit



Up to 200MHz (translating to a 10ns internal CPU clock cycle), a lower-cost 10K-series oscillator will work fine. Above that speed, a 100K-series oscillator should be used.

Due to internal chip circuitry, the test clock input signals **testClkIn_h,_l** should be pulled to the appropriate level using small resistors (100 ohms maximum). **testClkIn_h** should be pulled high (that is, to 3.3V through a small resistor) and **testClkIn_l** should be pulled low (to ground).

3.4 Unused Inputs

There are several inputs that are not used in a 21064-based system, but must be tied off either high or low. The following inputs should be pulled to 3.3V through a resistor:

- tagOk_h (unless using the tagOk function)
- tristate_l
- cont_l
- perfCntIn_h[1:0]

The following inputs should be pulled to ground:

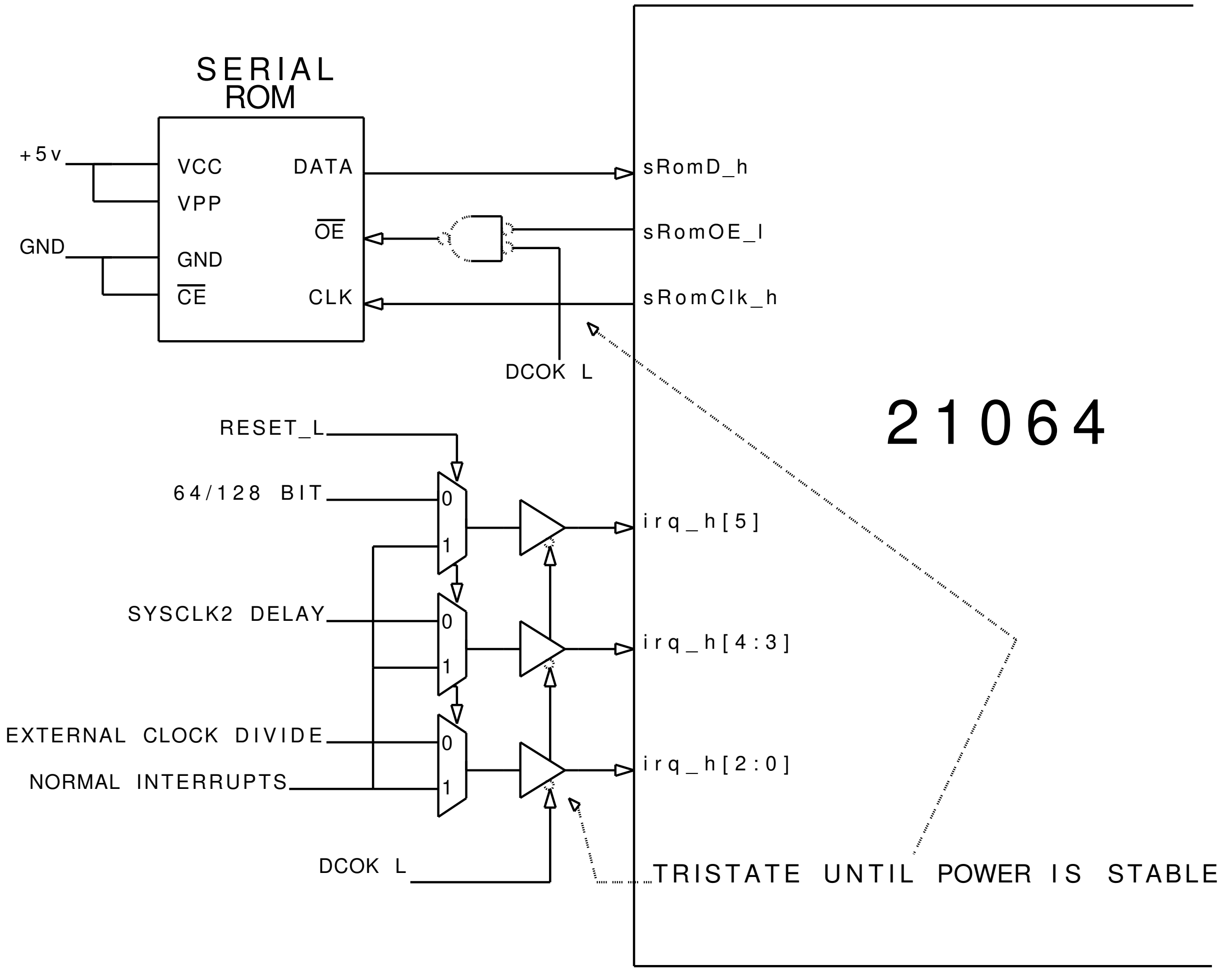
- tagOk_l (unless using the tagOk function)
- dWSel[0] (unless in 64-bit data bus mode)
- eclOut_h
- icMode_h[1:0]

The **tagOk_h,l** signals are used to stall the 21064 so that the Bcache can be controlled by the system logic. They are optimized for very high performance systems, and are not discussed in this document. The preliminary data sheet provides more details about the signals and their use. This application note discusses the simpler *holdReq_h* method for the system logic to take control of the Bcache (see Section 8.

4 Booting the 21064

The 21064 uses a flexible method to bootstrap the processor. Instead of always jumping to a fixed I/O address upon reset, the chip can load its initial I-stream from a compact serial ROM (SROM). As well, the configuration of the external interface is programmable by setting up certain input pins at reset time. Figure 7 shows how the serial ROM and the configuration inputs are used at reset time.

Figure 7: Serial ROM and Programmable Clock Inputs



While the 21064 is in reset mode, the interrupt input lines **irq_h[5:0]** are inspected to determine how the chip should configure the external interface logic. There are three configurable areas:

1. The 21064 can accommodate either a high-performance 128-bit external data bus or a lower-cost 64-bit data bus. **irq_h[5]** determines which of the two is selected, and is asserted high to choose the 128-bit mode. This application note describes the 21064 in 128-bit mode, but the preliminary data sheet provides more information about the differences.
2. The external interface runs synchronously to the external system clock, **sysClkOut1_h**. This external clock is generated from the internal clock, which can be divided by any value between 2 and 8 to form **sysClkOut1_h**. So, for example, the 21064 chip running at its nominal 6.6ns internal clock cycle time can be divided by 4 to allow an external interface to run at 26.4ns. **irq_h[2:0]** select the external interface division factor. Table 1 is a chart of the clock divisor decode.
3. The external interface logic is supplied two differential clocks from the 21064, **sysClkOut1_h,_l** and **sysClkOut2_h,_l**. Each external clock runs at the external cycle time selected above. **sysClkOut2** can also be delayed from **sysClkOut1** by a programmable value selected from **irq_h[4:3]**. The second clock can be delayed from 0 to 3 internal CPU clocks based upon this selection. Table 2 shows the delay times possible and their decode meaning.

Table 1: System Clock Divisor

irq_h[2]	irq_h[1]	irq_h[0]	Ratio
0	0	0	2
0	0	1	3
0	1	0	4
0	1	1	5
1	0	0	6
1	0	1	7
1	1	0	8
1	1	1	8

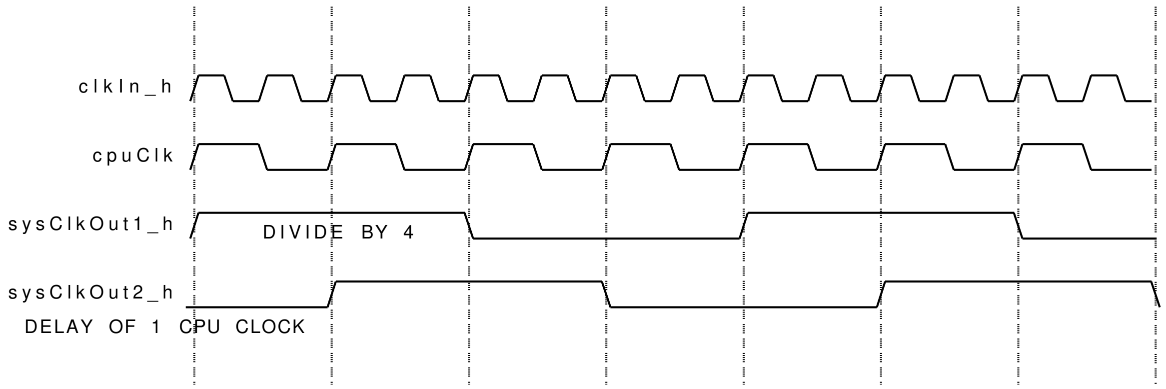
Table 2: System Clock Delay

irq_h[4]	irq_h[3]	Delay
0	0	0
0	1	1
1	0	2
1	1	3

Figure 8 shows how the clock configuration works. The input clock that is provided to the 21064 chip is divided by 2 in order to create the internal CPU clock. The CPU clock is the reference to all the other clocks that the chip outputs. In the example, the clock divisor is 4, so the system output clocks run at 1/4 of the internal CPU clock time. The figure shows that

sysClkOut2 has been delayed by 1 CPU clock from **sysClkOut1**. Since the external output clocks are differential, a two-phase clock is also available by using **sysClkOut1_h,_l**.

Figure 8: Example of 21064 Clock Configuration



When the **reset_l** signal de-asserts, the serial ROM is loaded into the processor Icache. The CPU controls the output enable and the clock for the ROM, and accepts the bit serial data. The preliminary data sheet provides details about the timing of the SROM control signals. After the SROM data has been loaded into the Icache, the processor jumps to location 0, which will hit inside the Icache. The SROM code is expected to perform chip and system initialization, preparing the system for external operation.

After the SROM code has been executed, it is assumed that the external interface is ready to supply I-stream data to the 21064 processor. The Bcache can be on or off at this point (in fact, there is no need to even have a Bcache if the user has no performance reason to include it). A general system might include a more complete boot/diagnostic ROM (BDROM) after the SROM has done its job.

Once the 21064 is executing in I-stream mode from an external interface, it expects full 32-byte fills. The normal data path of the 21064 is 128 bits (16 bytes), so two complete fill cycles are necessary to provide the 32 bytes of data. The BDROM code can be loaded and executed in several ways, though the suggested method is to move the BDROM code into RAM memory, then execute it from there. This can be easily handled by the serial ROM, which can read the BDROM byte by byte, pack it into appropriate memory words, move it into main memory, then jump to it in RAM.

5 Cache/memory Interface Details

The Bcache subsystem is carefully integrated into the 21064. It is expected that the Bcache SRAMs can be directly controlled by the 21064 pin bus, and that the Bcache data lines are connected to the 21064 data bus, as shown in Figure 2.

The rest of this description assumes that a Bcache does exist and is enabled. The case where a Bcache is not part of the data path is much simpler, and this same document can be used to understand the design of such a system by merely ignoring those sections dealing with the Bcache.

The Bcache is organized into 32-byte blocks or larger, with parity or ECC on 4-byte (32-bit) segments (no error detection is also an option). When the Bcache is enabled, the 21064 will generally probe it for each memory access (lock-related cycles are an exception). The tag and control SRAMs will first be enabled at the appropriate address, and if the probe finds a valid match the cycle will be finished without performing a main memory read or write cycle. The first 128-bit (16-byte) data segment will be read at the same time as the Bcache tag probe, and will be ready if the probe is successful. The 21064 will then read the second 128-bit segment. If the internal cache is enabled, the data is saved inside the chip. The preliminary data sheet provides a timing diagram of the 21064-controlled Bcache access cycles.

The Bcache is best utilized in writeback mode, which means that both reads and writes are normally serviced from the Bcache without external logic intervention. This implies that the Bcache has the only valid copy of a data block after it's been modified. The 21064 will manipulate the Bcache DIRTY bit to signify that the block has been written since it was initially read from memory. There is a method that the system logic can use to force non-writeback behavior, but its use is beyond the scope of this document. The preliminary data sheet discusses the **SHARED** Bcache bit in more detail.

5.1 Bcache Timing for 21064 Access

The Bcache timing is under complete control of the user through the BIU_CTL internal processor register (IPR). Figure 9 shows the layout of this register, which will normally be set up as part of the chip initialization code. The number of internal CPU cycles to allocate for Bcache reads and writes can be specified, along with the exact representation of where the Bcache write pulse will be asserted for Bcache writes.

Figure 9: BIU_CTL Internal Processor Register

```
6    3 3   3 3 3   2 2               1 1 1
3    6 5   2 1 0   8 7               3 2 1       8 7       4 3 2 1 0
+-----+-----+-+-----+-----------------+-+---------+---------+-+-+-+-+
|     |     | |     |                 | |         |         | | | | |
|  0  |     |0|     |BC_WE_CTL[15..1] |0|BC_WR_SPD|BC_RD_SPD|0| | | |
|     |     | |     |                 | |         |         | | | | |
+-----+-----+-+-----+-----------------+-+---------+---------+-+-+-+-+
         |       |                                             | | |
         |       |                                             | | +-> BC_EN
         |       |                                             | +---> ECC
         |       |                                             +-----> OE
         |       |
         |       +---------------------------------------------------> BC_SIZE
         |
         +-----------------------------------------------------------> BC_PA_DIS
```

Table 3: BIU Control Register

Field	Type	Description
BC_EN	WO,0	Bcache enable. When clear, this bit disables the Bcache. When the Bcache is disabled the 21064 does not probe the Bcache tag store for read and write references; it launches a request on cReq_h immediately.
ECC	WO	When this bit is set, the 21064 generates/expects ECC on the check_h pins. When this bit is clear the processor chip generates/expects parity on four of the check_h pins.
OE	WO,0	When this bit is set, the 21064 does not assert its chip enable pins during RAM write cycles, thus enabling these pins to be connected to the output enable pins of the cache RAMs.
BC_RD_SPD	WO	Bcache read speed. This field indicates to the BIU the read access time of the RAMs used to implement the off-chip Bcache, measured in CPU cycles. It should be written with a value equal to one less the read access time of the Bcache RAMs. Access times for reads must be in the range 16..3 CPU cycles, which means the values for the BC_RD_SPD field are in the range of 15..2. BC_RD_SPD are not initialized on reset and must be explicitly written before enabling the Bcache.
BC_WR_SPD	WO	Bcache write speed. This field indicates to the BIU the write cycle time of the RAMs used to implement the off-chip Bcache, measured in CPU cycles. It should be written with a value equal to one less the write cycle time of the Bcache RAMs. Access times for writes must be in the range 16..2 CPU cycles, which means the values for the BC_WR_SPD field are in the range of 15..1. BC_WR_SPD are not initialized on reset and must be explicitly written before enabling the Bcache.
BC_WE_CTL	WO	Bcache write enable control. This field is used to control the timing of the write enable and chip enable pins during writes into the data and tag control RAMs. It consists of 15 bits, where each bit determines the value placed on the write enable and chip enable pins during a given CPU cycle of the RAM write access. When a given bit of BC_WE_CTL is set, the write enable and chip enable pins are asserted during the corresponding CPU cycle of the RAM access. BC_WE_CTL[0] (bit 13 in BIU_CTL) corresponds to the second cycle of the write access, BC_WE_CTL[1] (bit 14 in BIU_CTL) to the third CPU cycle, and so on. The write enable pins will never be asserted in the first CPU cycle of a RAM write access. Unused bits in the BC_WE_CTL field must be written with zeros. BC_WE_CTL is not initialized on reset and must be explicitly written before enabling the Bcache.
BC_SIZE	WO	This field is used to indicate the size of the Bcache. BC_SIZE is not initialized on reset and must be explicitly written before enabling the Bcache. See Table 4 for the encodings.

Table 3 (Cont.): BIU Control Register

Field	Type	Description
BC_PA_DIS	WO	This 4-bit field may be used to prevent the CPU chip from using the Bcache to service reads and writes based upon the quadrant of physical address space which they reference. The correspondence between this bit field and the physical address space is shown in Table 5. When a read or write reference is presented to the 21064 the values of BC_PA_DIS, BC_ENA and physical address bits [33:32] together determine whether to attempt to use the Bcache to satisfy the reference. If the Bcache is not to be used for a given reference the chip does not probe the tag store, and makes the appropriate system request immediately. The value of BC_PA_DIS has NO impact on which portions of the physical address space may be cached in the primary caches. System components control this via the dRack field of the pin bus. BC_PA_DIS are not initialized by reset.

Table 4: BC_SIZE

BC_SIZE	Size
0 0 0	128 Kbytes
0 0 1	256 Kbytes
0 1 0	512 Kbytes
0 1 1	1 Mbytes
1 0 0	2 Mbytes
1 0 1	4 Mbytes
1 1 0	8 Mbytes

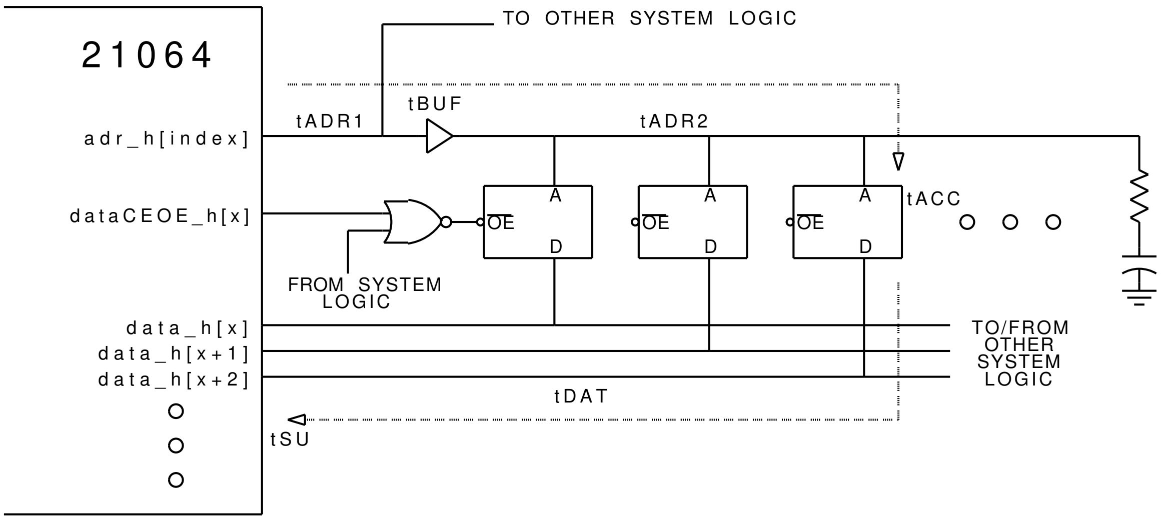
Table 5: BC_PA_DIS

BIU_CTL bits	Physical Address
[32]	PA[33..32] = 0
[33]	PA[33..32] = 1
[34]	PA[33..32] = 2
[35]	PA[33..32] = 3

5.1.1 Bcache Read Cycle

For a Bcache read cycle, the access/cycle time is determined by adding up the complete address or control path from the 21064 pin bus until the data is valid at the 21064 data bus pins. There is a 5ns setup requirement inside the 21064 on data reads, and this must also be considered. A system designed with the 21064 must provide access to the Bcache address and control signals from the module logic, so there is a NOR-type gate in the path. Furthermore, the 21064 output buffers are characterized driving a 40pf load, so any large fanout must be accomplished without exceeding this value. This usually means that buffers are added to the address and control paths.

Figure 10: Bcache Access Path for 21064



An example of a Bcache read access time calculation is provided here to clarify the steps. Figure 10 shows the general circuit assumed for this example. The address path drive signals will normally be treated as transmission lines in a real high-performance Bcache, so that is how they will be shown here. The termination scheme indicated in the figure assumes that your address buffer can drive a low impedance line to a proper level on the incident wave. If your driver cannot do this, then series termination should be used, with the implied increase in delay time due to the necessary reflection for a proper signal level. We will assume that the address buffer in the example has a specified propagation delay of 5ns. One of the address lines is assumed to be a fast, high-drive capability NOR-gate, and for our purposes it will be treated like the address buffer.

Many devices specify the maximum propagation delay with only one output switching, and in the case of an address buffer all the outputs might switch simultaneously. To account for this, extra buffer delay should be added to the assumed propagation delay through the device. For this example, we will assume that the 5ns buffer delay takes this into account.

All the calculations shown here are based upon the assumptions stated. The system or board designer is responsible for analyzing any particular implementation, and determining the correct delays and signal integrity issues. The purposes of this example are to show a general Bcache circuit that can be implemented with the 21064, and to explain how to program the IPR that controls the Bcache. Faster and slower systems can be built with the 21064 processor.

The SRAMs in our example have a specified access time of 20ns from address stable to data valid at their output pins. SRAM devices often have a faster specification from output enable to data valid, and it will be assumed that the *address* path, not the output enable path, is the critical one. The designer should ensure that this is true for any specific implementation.

The output enable path can be analyzed similarly to the address path. So the general components of delay for this calculation are:

```
tADR1 [delay from CPU to input of address buffer]
tBUF  [buffer gate delay]
tADR2 [address delay from buffer to SRAM inputs]
tACC  [SRAM access time from address valid to data valid]
tDAT  [data return path from SRAM to 21064 input pins]
tSU   [internal 21064 data setup time]
```

Figure 11: Timing Diagram for Bcache Read Access

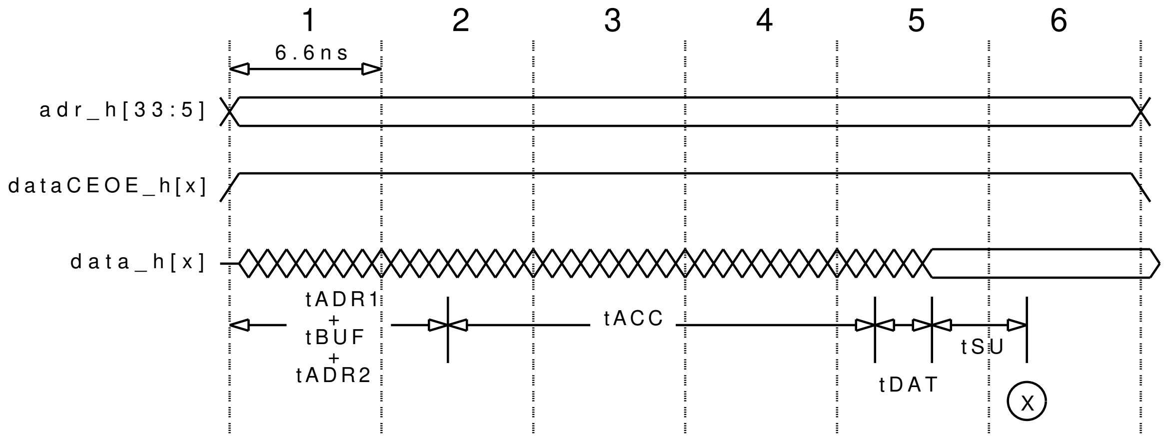


Figure 11 is a timing diagram showing the 21064 signals and their delay components. The valid data cannot be sampled until the point labeled "X" in the figure. The preliminary data sheet provides more detailed timing diagrams of the fast Bcache access path. The Bcache probe and each data read access would have the timing shown in Figure 11, and they are controlled by the same programmable BIU_CTL field.

The three unknown delay components are the address paths (tADR1, tADR2) and the data return path (tDAT). The data return path (tDAT) depends on the edge rate of the SRAM output, the length of the data line, and the other loads that are connected to the data line. As such, it is impossible to specify a "normal" delay time. For this exercise, it will be assumed to be 2ns.

The address delay path from the 21064 address output to the buffer (tADR1) is similar to the data path. It is unlikely to be a classical transmission line, due to the line length in relation to the edge rate of the 21064 output. However, there will likely be other loads on the address line, and the etch itself will cause a delay of around 160ps to 200ps per inch. For this example, tADR1 will be specified to be 2ns.

The path tADR2 needs a more classical transmission line analysis, since the buffers will have a fast switching time in relation to the line length. Even if the address drivers can switch the line to a proper level on the incident wave, the wave will propagate along the transmission line more slowly than if it was unloaded. Each SRAM will contribute some capacitance to the line, which will slow the wave down according to the formula:

```
tPL = tPD * SQRT(1+Ca/Co)
```

The term tPL is the loaded propagation delay per unit length, tPD is the propagation delay per unit length of the unloaded line, Ca is the added capacitance per unit length due to the SRAM inputs, and Co is the unloaded transmission line capacitance per unit length. It will be assumed for this example that it will take the wave 2ns to reach the last SRAM address input, where there will be no reflection. If the address driver cannot switch the line on the incident wave, a series termination scheme would be used instead, and the delay value would be higher.

So, the full trip from address valid at the 21064 output pin to data valid at the 21064 input pin (plus data setup) is:

```
 2ns   tADR1
 5ns   tBUF
 2ns   tADR2
20ns   tACC
 2ns   tDAT
 5ns   tSU
-------
36ns
```

The numbers above are only for this example. If the designer uses different buffers, or splits the address drivers differently, or uses drivers that cannot switch the low impedance line on the incident wave, the analysis would change accordingly. We will assume that the 21064 is using an internal cycle time of 6.6ns, which means that the chip must allocate 6 cycles for the Bcache read given the conditions specified. This is programmed into the BIU_CTL register by setting the BC_RD_SPD field to 5, since the actual cycle count is one more than the one specified in the register. This value will work for any round trip delay that is less than or equal to 39.6ns.

It should be noted that using SRAMs with an access time of 17ns would reduce the number of internal CPU cycles to 5, assuming that everything else remained constant.

5.1.2 Bcache Write Cycle

A fast CPU-activated Bcache write cycle can be analyzed similarly. The BC_WR_SPD field in the BIU_CTL register should be programmed so that the SRAM write cycle will finish, and the BC_WE_CTL field should place the write pulse so that the timing and width do not violate the SRAM specifications.

An example of this calculation is provided here. Figure 12 shows the circuit that is assumed for the Bcache write path.

Figure 13 shows a timing diagram of the write path signals as viewed from the 21064. The preliminary data sheet provides a detailed timing diagram of a fast Bcache write access. The tag probe follows the timing for a fast Bcache read, and each write access follows the timing as shown in Figure 13. The write pulse cannot assert until point "X" in the figure, and it cannot de-assert until point "Y" in the figure. The cycle cannot end until point "Z" in the figure.

Figure 12: Cache Write Path for 21064

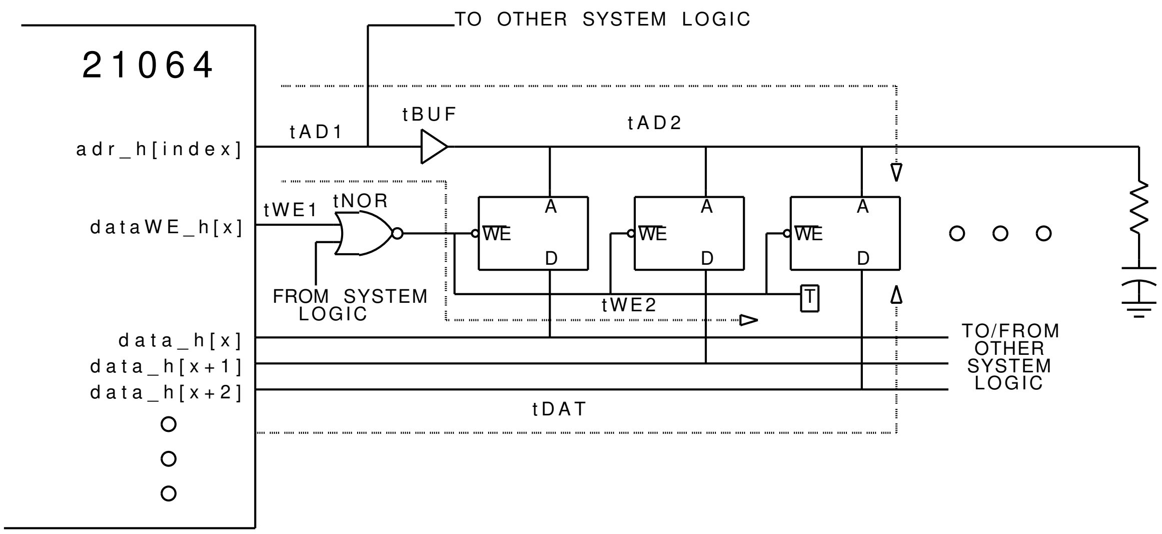
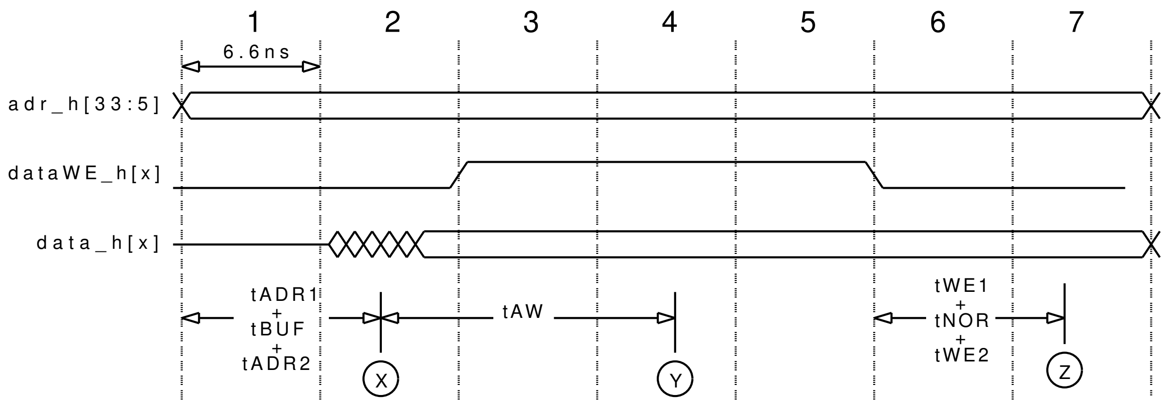


Figure 13: Timing Diagram for Bcache Write Access



It will be specified for this example that the minimum write pulse width for the SRAM (tWM) is 15ns. The 21064 can have 1.5ns of skew between the rising and falling edges of the pulse it will generate. Furthermore, although the rise and fall delays through the NOR gate in the figure should be close, some skew must be added to account for:

- Potential input threshold differences inside the SRAM
- Differences that result in a rise propagation delay that is different than the fall propagation delay

For the purposes of this example, we will add 2ns of skew between the rising and falling edges of the write pulse (1.5 for the 21064, and 0.5 for the logic and threshold differences). The following SRAM specifications will be used for this example:

```
tWC = 20ns  [Write cycle time]
tWP = 15ns  [Write pulse width]
tDW =  8ns  [Data setup time to write pulse de-assertion]
tDH =  0ns  [Data hold time from write pulse de-assertion]
tAW = 15ns  [Address setup time to write pulse de-assertion]
tWR =  0ns  [Address hold time from write pulse de-assertion]
tAS =  0ns  [Address setup time to write pulse assertion]
```

The above specifications are only a subset of the total device specifications for a real device, and are used to show the general technique used in determining how to program the BIU_CTL IPR. Designers should closely analyze their own systems, including the device support logic, etch paths, and RAM specifications, in order to determine exactly which paths are critical.

The first BIU_CTL field to calculate is the BC_WE_CTL, which determines where the write enable pulse will be asserted. The field is 15 bits wide, and each bit represents an internal CPU cycle that will assert the write enable pulse (starting with the second cycle, since the first cycle will never drive the write enable pulse).

The address delay calculation is similar to the read case, and it should be added to the address setup time as follows:

```
 2ns   tADR1
 5ns   tBUF
 2ns   tADR2
15ns   tAW for SRAM
-------
24ns
```

The earliest that the write pulse can be de-asserted is then 24ns from the start of the write cycle, based upon the address setup requirement.

There are two types of "data" that need setup and hold time for the write cycle. The actual Bcache data is the first type, and the tag control inputs (VALID, DIRTY, SHARED, and PARITY) are the second type. The 21064 drives the tag control inputs one CPU cycle later than the actual data, and we will assume that they are the critical path. The chip will provide stable data at most 2.9ns after the nominal edge that drives the data (in this case, tag control) lines. We will assume that the data will take 2ns to get to the SRAMs and be stable. If the CPU clock cycle is 6.6ns, then the earliest that the write pulse can de-assert is calculated as follows:

```
 6.6ns   [1 CPU clock cycle]
 2.9ns   [21064 data stable time]
 2.0ns   tDAT
 8.0ns   tDW for SRAM
-------
19.5ns
```

It would appear that in this example the address path is the critical one, and the write pulse cannot de-assert until 24ns after the start of the write cycle. The minimum pulse width is specified to be 15ns, which must be extended to (15+2=) 17ns to account for the pulse width skew in the 21064 and the external logic. At an internal 6.6ns CPU cycle time, 3 cycles must be used for the write pulse.

Since the earliest that the write pulse can de-assert is 24ns after the start of the write cycle, the latest that it can assert (in order to meet that de-assertion time) is (24-17=) 7ns after the cycle start. We have specified here that the write pulse cannot assert until the address is stable (tAS), and this will put a bound on how early the write pulse is asserted. It was determined previously that the address will reach the last SRAM (tADR1+tBUF+tADR2=2+5+2=) 9ns after the start of the cycle. Since there is also 1.5ns of skew between the address signal and the write pulse signal coming from the 21064, the real minimum time is (9+1.5=) 10.5ns from the cycle start.

So, the earliest that the 21064 can assert the write pulse is 13.2ns into the Bcache write (that's the beginning of the 3rd CPU cycle). The write pulse should then start at the 3rd CPU cycle and extend until the end of the 5th cycle. The BC_WE_CTL field should be programmed to be 000000000001110. This means that the write pulse will remain asserted until (6.6ns*5=) 33ns into the Bcache write, which puts it after the 24ns limit previously calculated.

The other programmable field of interest in the BIU_CTL is the BC_WR_SPD field, which determines the entire write cycle time. The write pulse itself is de-asserted at the end of the 5th CPU cycle into the Bcache write in this example, which means it nominally de-asserts (6.6*5=) 33ns from the start of the cycle. It might be 1.5ns later than that due to 21064 output skew. There is also a NOR gate in the path (tNOR), and some wire travel time associated with the signal (tWE1 and tWE2).

There are three components of delay for the write enable pulse. The two write delay components (tWE1 and tWE2) might or might not be transmission lines. Figure 12 implies that tWE1 is not a transmission line and tWE2 is, with parallel termination. The module designer should analyze the particular implementation to see what the correct configuration should be, and if one of them is a transmission line it should be terminated appropriately (this analysis is similar to the address calculation in the previous section).

We will assume that tWE1 is 1ns, tNOR is 5ns, and tWE2 is 2ns for this example. So, the latest that the write pulse can de-assert at the last SRAM (and thus the earliest that the cycle can end) is:

```
33.0ns   [nominal write pulse de-assertion from start of write]
 1.5ns   [21064 skew from nominal edge]
 1.0ns   tWE1
 5.0ns   tNOR
 2.0ns   tWE2
-------
42.5ns
```

At a 6.6ns cycle time this translates to 7 cycles, so the value of 6 should be programmed into the BC_WR_SPD field (since this value is always 1 less than the actual write cycle time). The nominal write cycle speed will be 46.2ns for this example. As with the read cycle, it will be noted here that if the write enable pulse requirement was shorter (say 11ns rather than 15ns), the fast Bcache write could be reduced to 6 cycles.

5.2 Bcache Miss and External Request

An initial Bcache fill operation is executed when the 21064 attempts to read or write a block that misses in the Bcache (the write fill operation assumes a write-allocate Bcache policy). The miss can be caused for several reasons:

1. The Bcache block for that index is not valid
2. The Bcache block for that index is valid, but the tag misses

The first scenario above is the simplest, and will be discussed first. When a Bcache probe results in a miss, an external READ_BLOCK or WRITE_BLOCK operation is initiated by the 21064 external interface logic. The READ_BLOCK and WRITE_BLOCK external cycles are the most basic method of transferring data between the 21064 and the system, and are discussed in some detail in this document. The preliminary data sheet provides more details about other command types.

The command is initiated when the 21064 places the appropriate code on the **cReq_h[2:0]** lines during the rising edge of **sysClkOut1_h**. Timing for external cycles is synchronous to **sysClkOut1_h**, and all setup and hold times are referenced to the rising edge of this clock. The address, control, and data signals all change simultaneously with **sysClkOut1_h**, and therefore cannot be sampled on that same edge. In general this is only a concern for those lines that are used to determine if a cycle should begin, such as the request lines **cReq_h[2:0]** (the **holdAck_h** line is also in this category, as discussed later). A delayed version of **cReq_h[2:0]**, perhaps sampled by **sysClkOut2_h**, should be used to feed any state machines that run on **sysClkOut1_h** and use the request lines.

Figure 14: External Cycle

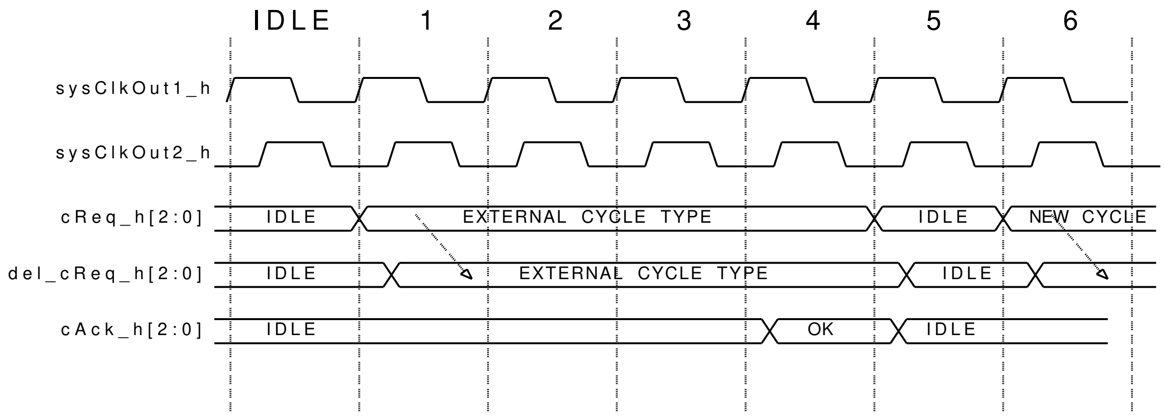


Figure 14 shows this relationship. The 21064 places the external cycle type on the **cReq_h[2:0]** lines at the start of cycle 1 in the figure. **sysClkOut2_h** is used to sample the request lines, and the system logic uses this delayed version to start its state machines at the start of cycle 2. After the external logic has performed the appropriate function, it changes the **cAck_h[2:0]** lines, which are sampled by the 21064 at the start of cycle 5. The CPU removes the request lines at that same time, and could start a Bcache access immediately (at the start of cycle 5). The earliest that the CPU can start another external cycle is one system clock cycle later, at the start of cycle 6 (as shown).

It is assumed here that the Bcache block is invalid, but the external logic would have no way to know that. So, the external logic must have some way to determine if the current Bcache block occupant needs to be written back to the main memory. One method to do this is to have the system logic perform its own Bcache tag probe. Only the VALID and DIRTY bits need to be inspected, so the external logic probe does not have to wait the entire time necessary to compare the tag address field in the Bcache.

A critical path in this external logic probe is the SRAM output enable circuitry. The 21064 leaves the Bcache RAMs disabled after its own probe, and it's up to the external logic to drive the output enable again in order to inspect the VALID and DIRTY bits. One way to do this is to allow an early version of the **cReq_h[2:0]** signals to turn on the SRAM output enables by default, assuming that a probe will be necessary. For those cycles where the external logic later needs to write the Bcache, another logic path is necessary to turn the output enable back off. The de-assertion path is not time-critical, but does need to be implemented for cache fill operations.

Figure 15: Tag Control Probe Before External Cycle

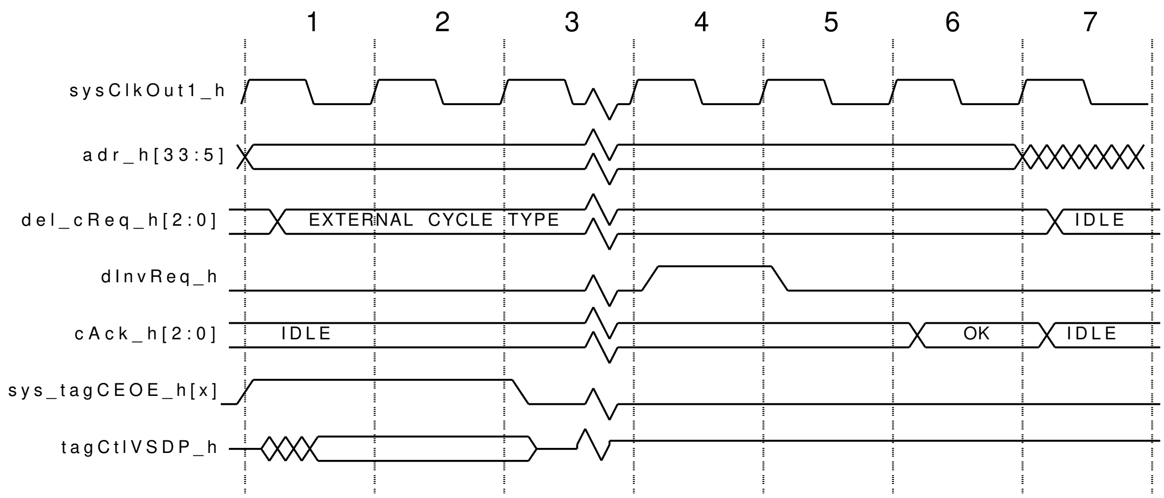


Figure 15 shows the timing for the entire cycle, including the tag control check. The figure shows the delayed **cReq_h[2:0]** lines changing state at the start of cycle 1, and the tag probe occurring during that cycle. The signal **sys_tagCEOE_h[x]** is the system logic version of the 21064 **tagCEOE_h[x]** signal. It is the other input to the NOR gate shown in Figure 3. That nomenclature will be used throughout this application note.

In this case, it was assumed that the Bcache block is invalid, so no victim write needs to be performed. Section 5.5 explains the details of a victim write. If the Bcache probe found that the block was valid but not dirty (that is, it had not been modified since being read from main memory), then the outcome is the same. In both cases, the block can safely be invalidated without a victim write.

Figure 15 shows the tag inspection being implemented in one cycle, so that the read command can start at the beginning of cycle 2. This might not be possible on any particular implementation, and must be carefully analyzed to ensure that the data will be stable when the clock asserts in the system control logic.

The external cycle (READ_BLOCK or WRITE_BLOCK) will overwrite the data in the Bcache, and will assert the **dInvReq_h** signal if appropriate during the fill, so the internal Dcache block will be invalidated later. The lower address bits are directly connected to the **iAdr_h[12:5]** invalidate address input lines in this example, and that will ensure that the correct cache block will be invalidated. Some implementations might want better control over the invalidate bus, and must ensure that the **iAdr_h** lines accurately reflect the lower index value on the asserting edge of **sysClkOut1_h** that samples **dInvReq_h**.

5.3 Read Block Request

If the external cycle is a READ_BLOCK, a 32-byte block of memory information is returned to the 21064. The external logic has complete control of the 21064 pin bus during the transfer. The data is returned to the 21064 and simultaneously loaded into the Bcache. It is the external logic that writes the data into the Bcache during the read cycle, *not* the 21064.

The minimum amount of data that can be written to the Bcache is 32 bytes, but the system logic controls the Bcache until the **cAck_h[2:0]** lines are changed from their IDLE state. As such, it can load and validate more than that if the system designer believes prefetching more blocks is appropriate. Any prefetching must be done in 32-byte increments.

The external logic is responsible for loading the tag address and the tag control fields of the Bcache (with correct parity on both) along with the data. The tag control field should be written as VALID and CLEAN.

Figure 16: Tag Access and Write Circuit

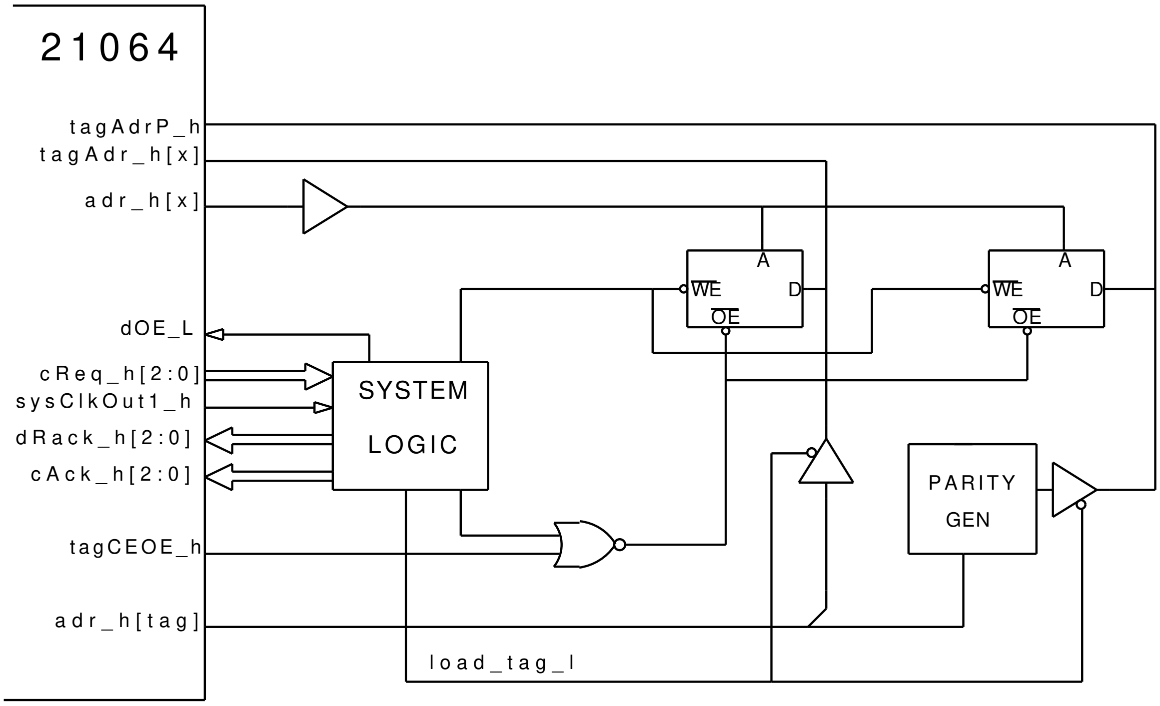
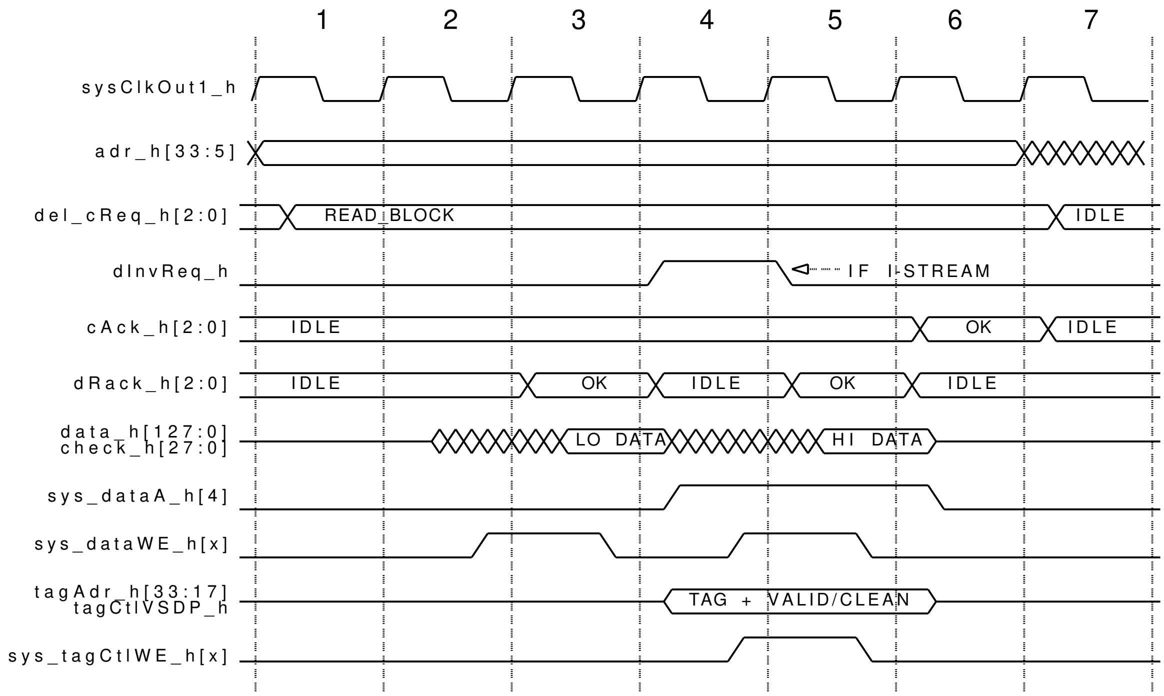


Figure 16 shows an example of the logic that is expected for tag address control. One of the **tagAdr_h** lines is shown connected to its Bcache RAM. All the tag address lines in use by the implementation go to the parity generator. The **tagAdr_h[tag]** signals and the **tagAdrP_h** parity lines are all driven by tristate buffers. On probe reads, the SRAM output enable allows the Bcache to drive the signals, where they are compared by the 21064 or the system logic. On a fill operation, the **load_tag_l** signal causes the Bcache tag RAMs to be loaded with the upper address bits. Notice that the RAM write enable input is not connected to the 21064, since the processor never writes them.

As each data word is returned to the 21064 the **dRack_h[2:0]** field is changed from IDLE to non-IDLE. Normally, the non-IDLE state will be OK, which instructs the 21064 to both check the ECC (or parity) on the returned data and cache the data internally. The preliminary data sheet provides more information on the **dRack_h[2:0]** field. Figure 17 is a timing diagram for a READ_BLOCK data transfer, showing the 21064 control signals.

The data in the example is assumed to be ready at the start of cycles 4 and 6, but in another implementation the data might be ready before or after that time. The **dRack_h[2:0]** lines should change to the non-IDLE state whenever the data is ready, with enough setup time so that they are sensed by the 21064 at the assertion of **sysClkOut1_h**. The **cAck_h[2:0]** lines can also change to their non-IDLE state (signifying the end of the cycle) during the last **dRack_h[2:0]** data phase if desired.

Figure 17: Timing Diagram of READ_BLOCK Cycle



The timing of the Bcache write signal might be tight in relation to the data arriving from the memory. If the memory is a DRAM array, for example, the CAS signal should be de-asserted as quickly as possible after the DRAM data is stable in order to start the next memory access during a page mode read. The Bcache, however, might need the data held stable. Using a bidirectional clocked memory data transceiver, as shown in Figure 2, can help in some cases.

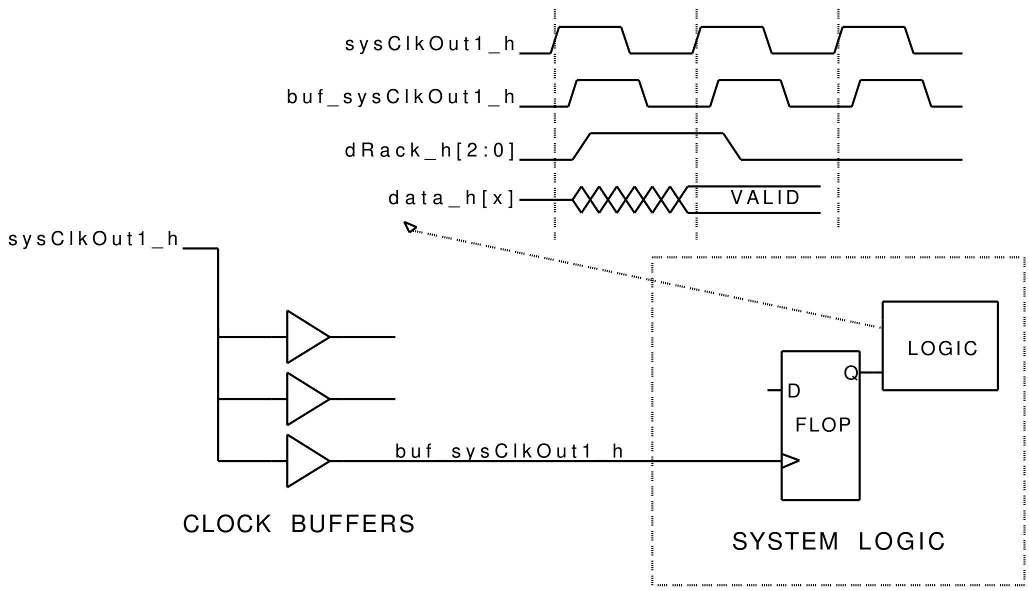
Figure 17 shows the **dInvReq_h** signal asserting, which will invalidate the internal Dcache block corresponding to the lower index bits in the address. This is only needed if the external data fetch is for I-stream (indicated by a false **cWMask_h[2]** on READ_BLOCK cycles), and if the internal Dcache is being kept as a subset of the Bcache. The block that is being filled into the Bcache might be in the Dcache, and it will not otherwise be invalidated on an I-stream fetch.

The 21064 can potentially drive its own Bcache control signals a few CPU cycles into the external cycle. As such, the Bcache SRAMs might still be driving the data bus as the external cycle starts. On a read cycle, the system logic might turn on its own data transceivers early in the access, and should be aware that a system cycle should be allowed before this is done. This eliminates any tristate overlap between the SRAMs and the data transceiver.

For a system without a Bcache, the 21064 signals would be the same as Figure 17, but none of the Bcache related lines would be asserted by the system logic. If the system has a Bcache but it is not enabled, the external system logic needs to have some mechanism to turn off the Bcache fill logic, since the 21064 does not broadcast its internal Bcache enable signal to the external pin bus.

If the read cycle is to an area of memory that has been defined as I/O, it is likely that another bus is involved with the transfer. In this case, the timing is also similar, and the Bcache control signals are also not asserted. A further modification in this case might be to change the **dRack_h[2:0]** field to indicate that no error checking be performed and that the data should not be loaded into the internal chip Dcache either.

Figure 18: Clock Skew From System to 21064

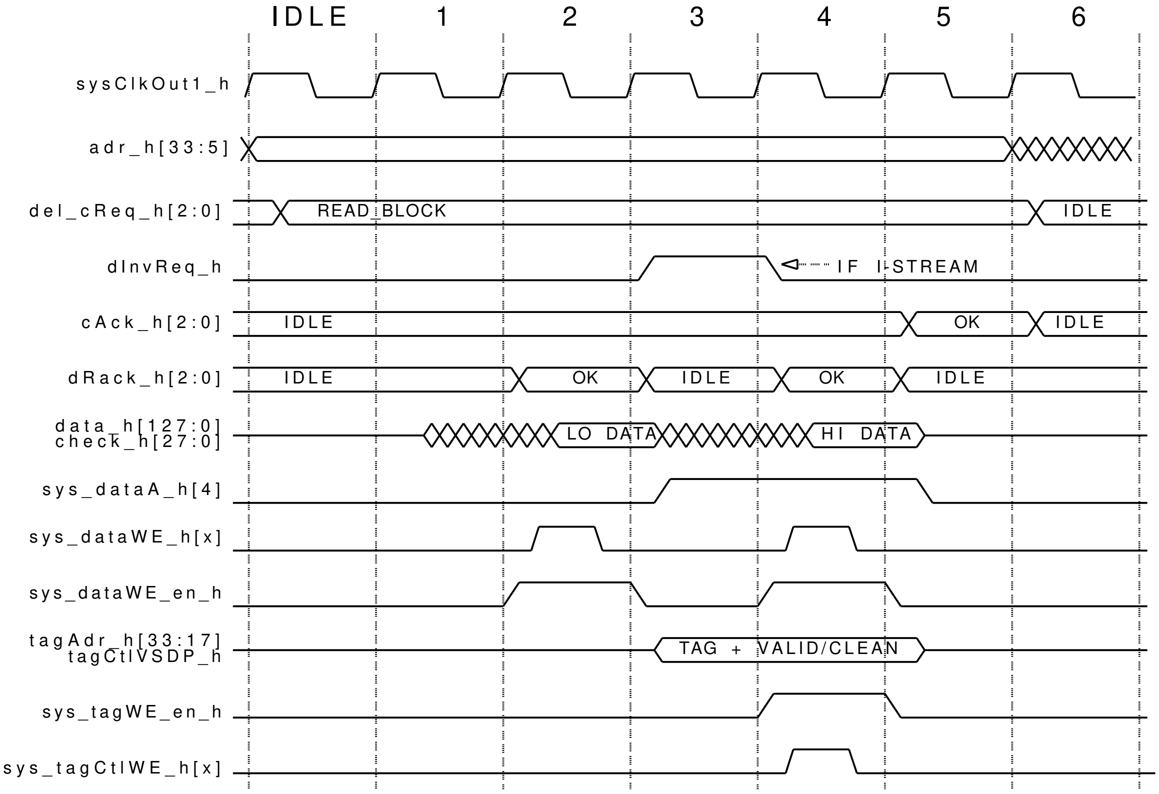


The 21064 system clocks, such as **sysClkOut1_h**, are specified to drive only 40pf. Because of this, clock buffers will normally be used to drive the system logic. The clock buffers add skew between the 21064 and the system logic. Figure 18 shows a timing diagram and a small circuit section that might be used to create the signals in the diagram. The buffered version of the system clock **buf_sysClkOut1_h** drives the system state machines that eventually cause the **data_h** lines to be valid at the 21064 input pins.

The **data_h** must be setup at least 3.5ns before the assertion of **sysClkOut1_h**. In this example, the delay added by the buffer must be added to that setup time, since the 21064 sees its reference clock some time before the system logic. This delay should include the entire path for the buffered clocks, including wire delay, device propagation delay, simultaneous switching increases, transmission line effects, etc. The example in Figure 18 shows only one instance of this consideration. Others must be analyzed based upon the implementation.

It should be noted here that the skew helps signals like **dRack_h[2:0]** and **cAck_h[2:0]**, since they can be asserted on the system logic version of the clock and meet both the setup and hold times in reference to the 21064.

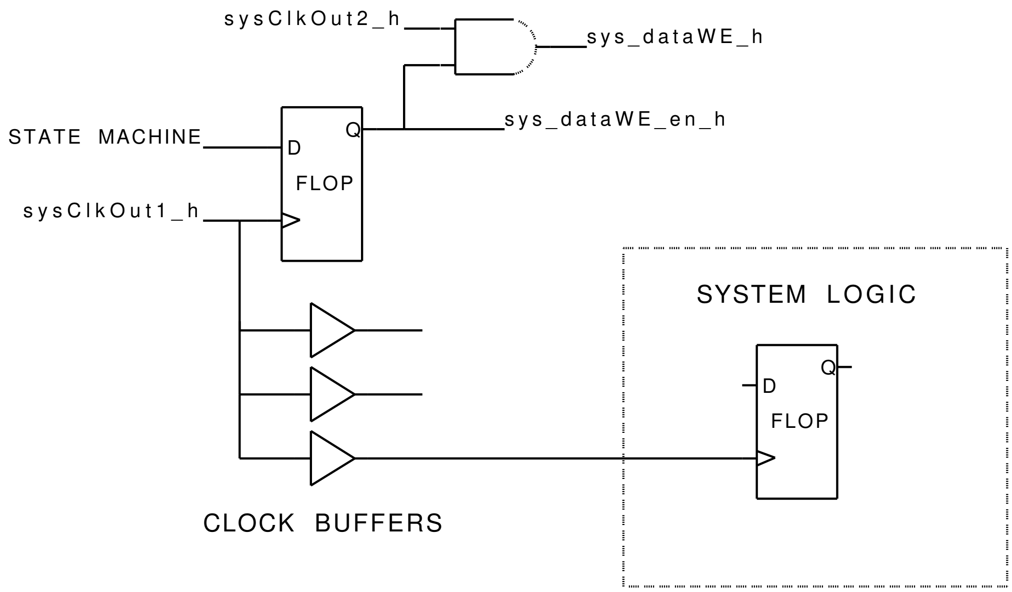
Figure 19: READ_BLOCK Cycle with Write Pulse



If the external timing allows it, a write pulse can be created by delaying **sysClkOut2_h** by 1 CPU cycle, and using **sysClkOut1_h** to create an enable signal for it. Figure 19 shows a READ_BLOCK cycle with a cache fill that uses a write pulse to load the Bcache. The signal **sys_dataWE_en_h** enables **sysClkOut2_h** when the Bcache needs to be written.

Figure 20 is an example of how the write pulse can be created, showing the circuit paths of interest. The clock buffers are shown that are expected to drive the system logic, in part to show that skew must be carefully considered if a write pulse-like scheme is attempted. If the clock buffers add enough delay to the path, and the delayed version of the clock is used to create the **sys_dataWE_en_h** signal, the leading edge of the enable can overlap with **sysClkOut2_h**. To prevent this from happening, a non-buffered version of **sysClkOut1_h** might be used to create **sys_dataWE_en_h**. The same argument applies to the tag control write pulse.

Figure 20: Write Pulse Circuit



5.4 Write Block Request

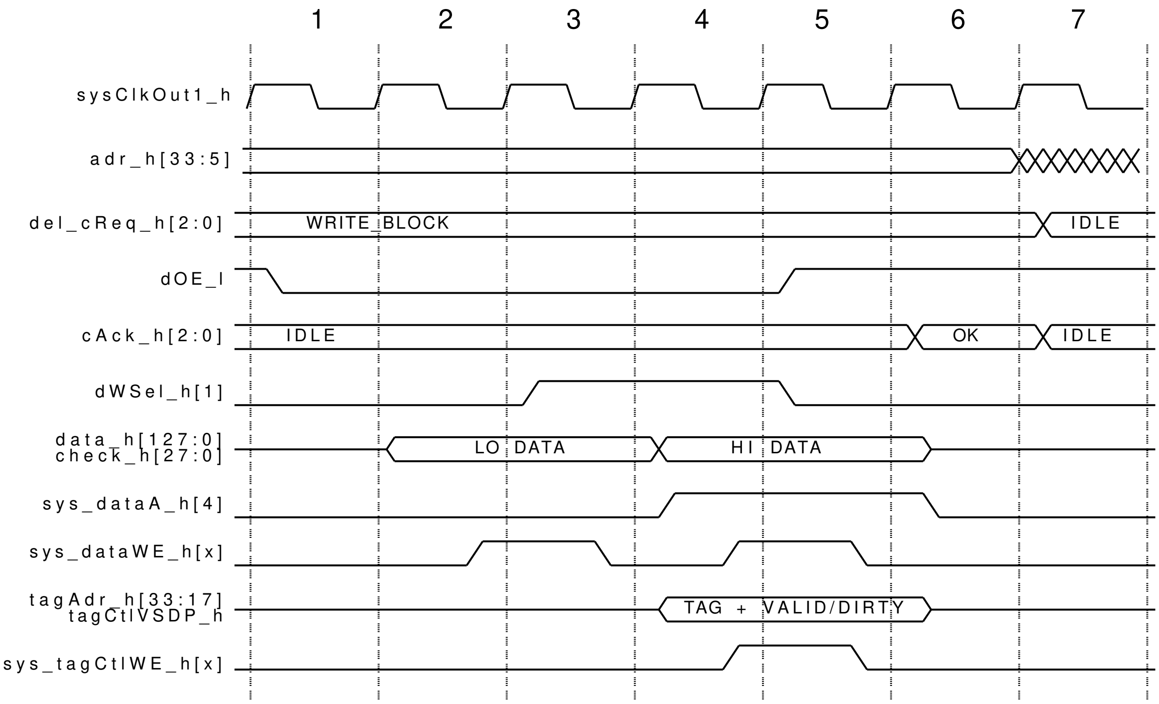
If the external cycle is a WRITE_BLOCK, the system logic must perform a different set of functions. The initial tag probe must still be done by the external logic, and we are still assuming here that the current block is either not valid, or it is valid but not dirty (no victim write needed).

If we assume that the Bcache is used as a writeback cache (the normal mode), and that the design is using a write-allocate Bcache policy, then the write data should go into the Bcache, even though an external WRITE_BLOCK cycle is being executed. The most reasonable way to accomplish this is to read the entire block from memory into the Bcache, then write the masked 8-byte into that same Bcache block. For systems without a Bcache, the external memory should be writable on 4-byte (32-bit) boundaries, since the Bcache merge cannot be performed.

The 21064 is attempting to perform a WRITE_BLOCK cycle in this case, and doesn't even know about the memory read cycle. The **dRack_h[2:0]** and **cAck_h[2:0]** signals should remain IDLE throughout the read transfer. After the read has been accomplished and the main memory data is now in the Bcache block, the system logic should cycle the 21064 through its write data by using the **dWSel_h[1]** line. The 21064 input signal **dOE_l** is used to instruct the chip to drive the data lines for the write portion of the cycle. Only the masked 4-byte segments should have their write enable inputs asserted during the cycle, based upon the **cWMask_h[7:0]** signals.

After the entire read and write cycle have been finished, the tag control should be written as VALID and DIRTY, and the tag address should be written with the correct upper address bits. Figure 21 shows the Bcache write portion of the WRITE_BLOCK cycle. The read

Figure 21: Timing Diagram of WRITE_BLOCK Cycle



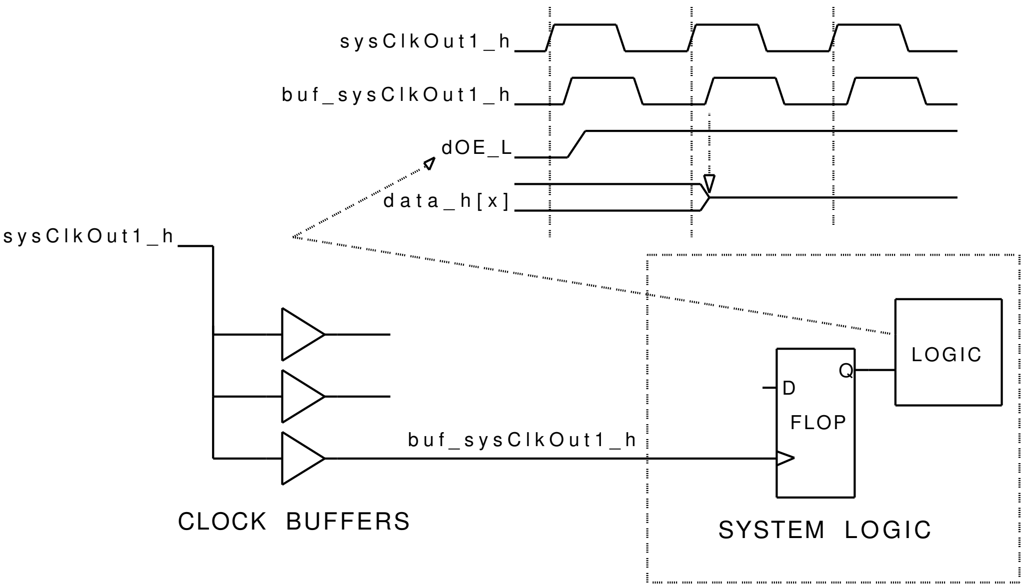
portion looks like Figure 17, except that the CPU acknowledge signals should not be changed from IDLE.

Note when the data actually changes relative to the signals **dWSel_h** and **dOE_l**. All the signals are synchronous to the leading edge of **sysClkOut1_h**, so the inputs are not acted upon until the next system clock edge. The end of the external write in Figure 21 is the start of cycle 7, at which time the 21064 will remove the address and potentially start the next Bcache probe.

There are several optimizations that can be made on the write cycle:

1. The **cWMask_h[7:0]** signals can be inspected, and if they are all set the read portion of the cycle does not have to be performed. In this case, every byte will be written anyway, so the Bcache write cycle can be performed from the start.
2. The tag Bcache RAMs don't have to be written on both the read and write portions of the cycle. It may turn out to be simpler to do it during the read cycle so that it is the same as a normal read, but it is under control of the designer.
3. Both 128-bit data segments don't need to be written if the lower mask bits show that there are no 4-byte segments enabled. The signal **dWSel_h[1]** can be asserted earlier to write the upper 128 bits only. If both segments *are* written, however, the lower address must be written before the upper address (as shown in Figure 21).

Figure 22: Clock Skew From System to 21064 for Write



As with the read cycle, the write cycle must take into account the clock skew between the 21064 and the system logic. Figure 22 shows an example of a potential problem. The 21064 signal **dOE_l** is asserted by the system logic to instruct the chip to drive the **data_h** lines during the write cycle. But **dOE_l** is sampled by the chip on the earlier, unbuffered version of **sysClkOut1_h**. In Figure 22, the data is removed on the asserting edge of **sysClkOut1_h**, which might be too soon. If the system logic uses its version of **buf_sysClkOut1_h** to sample the write data, then it should cause **dOE_l** to remain asserted low one extra cycle to accommodate the clock skew. This same argument applies to **dWSel_h[1]**.

5.5 Victim Write

The second possibility for the original Bcache miss is that the data currently occupying the Bcache block is VALID and DIRTY, but the upper address bits do not match the tag address. The 21064 will go to the external logic with a READ_BLOCK or WRITE_BLOCK, just as in the previous description. When the external logic does the Bcache VALID/DIRTY probe, however, the outcome is different. Since the data in the Bcache block has been modified since it was read from the main memory, it must be written back to memory before the external read or write cycle can continue. The act of writing the block back to memory is called a victim write.

The external control logic for a victim write is straightforward. The 128-bit data segments are read from the Bcache, and the data is sent to the external memory. After the victim is safely back in memory, the READ_BLOCK or WRITE_BLOCK is performed, exactly as described in the previous sections. Some time during the entire cycle (including the victim write and subsequent read or write cycle), the **dInvReq_h** signal should be asserted to invalidate the internal Dcache block for that index.

Figure 23: Timing Diagram of Victim Write Cycle

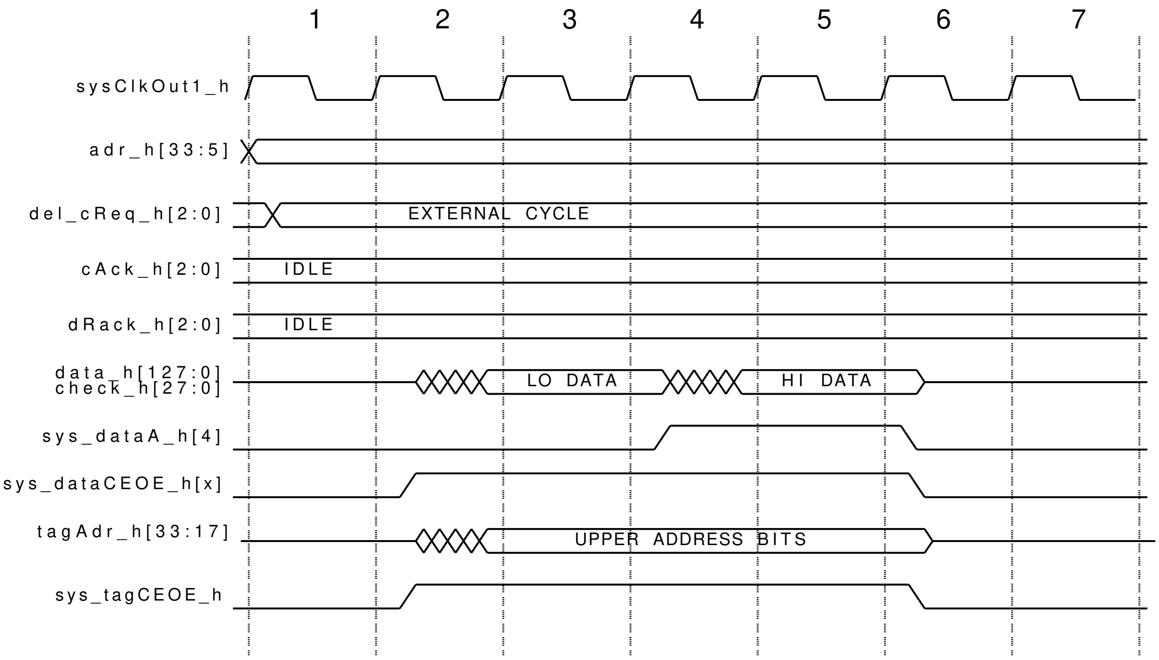


Figure 24: Address MUX for Victim Write

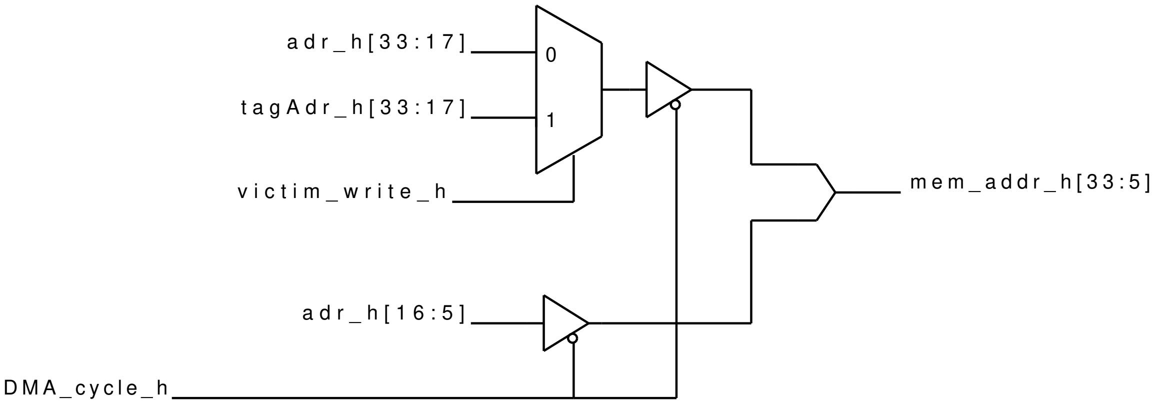


Figure 23 shows the victim write cycle. The tag address is used as the high memory address bits for the write, so the **sys_tagCEOE_h** signal is asserted to enable their outputs. The Bcache data RAMs are enabled, and each data segment is selected in turn by **sys_dataA_h[4]**. In this example, two cycles are necessary for the main memory to be written. If the memory is slower, more cycles should be allocated. At the start of cycle 7 the actual read or write cycle would proceed.

A MUX gate is needed to choose between the normal 21064 memory write and the victim write, where the upper address bits are taken from the tag field of the Bcache. Figure 24 shows the expected circuit. Normally, the MUX selects the **adr_h** lines, but during victim write cycles the **tagAdr_h** lines are chosen as the memory address. Figure 24 also shows that the entire address bus should have the ability to tristate for DMA access. During DMA transfers, the 21064 is forced off the address lines and the external logic controls the entire address. The MUX and tristatable gate can be one physical device.

The signals **victim_write_h** and **DMA_cycle_h** are expected to be created by the system logic. They do not come from the 21064. The tag address field in Figure 24 is shown for the smallest Bcache size. Other Bcache sizes will have different relative widths for the tag and index fields.

For high performance systems, a victim queue (or silo) is an option. Instead of writing the victim and reading the new data word serially, the Bcache and the memory can be read simultaneously. The information in the Bcache can be stored in a silo while the memory data is loaded into the Bcache. The silo can then be used to write the previous Bcache contents to memory.

5.6 Non-cacheable Memory Write

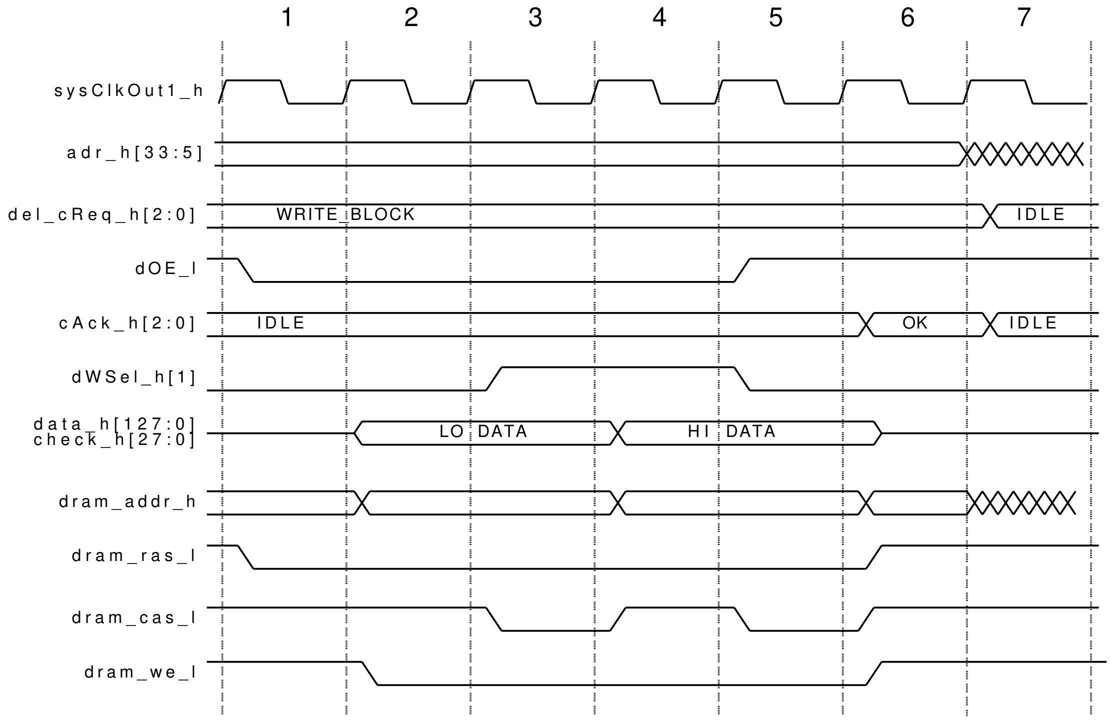
There might be non-cacheable memory space in your 21064 system design. When that area is a write target, the data should bypass the Bcache and be written directly to the system memory. If non-cacheable memory is included in the system, it is best to make it writable on 4-byte segments. Otherwise, a full read/modify/write cycle will be needed to store non-fully masked data.

A memory write on a system that allows masking on 4-byte segments is only a minor variant on the victim write function. The difference is that the information to be written to memory is coming from the 21064 rather than the Bcache. The Bcache is not invoked at all in this situation, and the **dWSEl_h[1]** signal is used to instruct the 21064 about which 128-bit data segment to provide.

Figure 25 shows the timing for such a write cycle. In this example, a more complete memory control flow is shown. It is assumed that the memory is a DRAM array, and a representative set of memory control signals are provided. The designer should work out the exact timing on a particular implementation in order to ensure that the memory parts are accessed within specification.

The **adr_h** lines should be stable at the start of the cycle, since they are changed by the 21064 before the cycle is started. If the DRAM address MUX points to the row address by default, the memory can be RASed at the start of the cycle. At the end of the cycle, the DRAM RAS precharge time must be accounted for. The 21064 will allow at least one idle cycle after it senses **cAck_h** as non-IDLE before it will start the next external command. In the example, RAS de-asserts at the start of cycle 6, which means that it cannot re-assert until the start of cycle 8. The changing of **cAck_h** so that it is sensed at the start of cycle 7 meets the RAS precharge time for the part in this implementation.

Figure 25: Timing Diagram of Direct Memory Write Cycle



6 Load Locked and Store Conditional

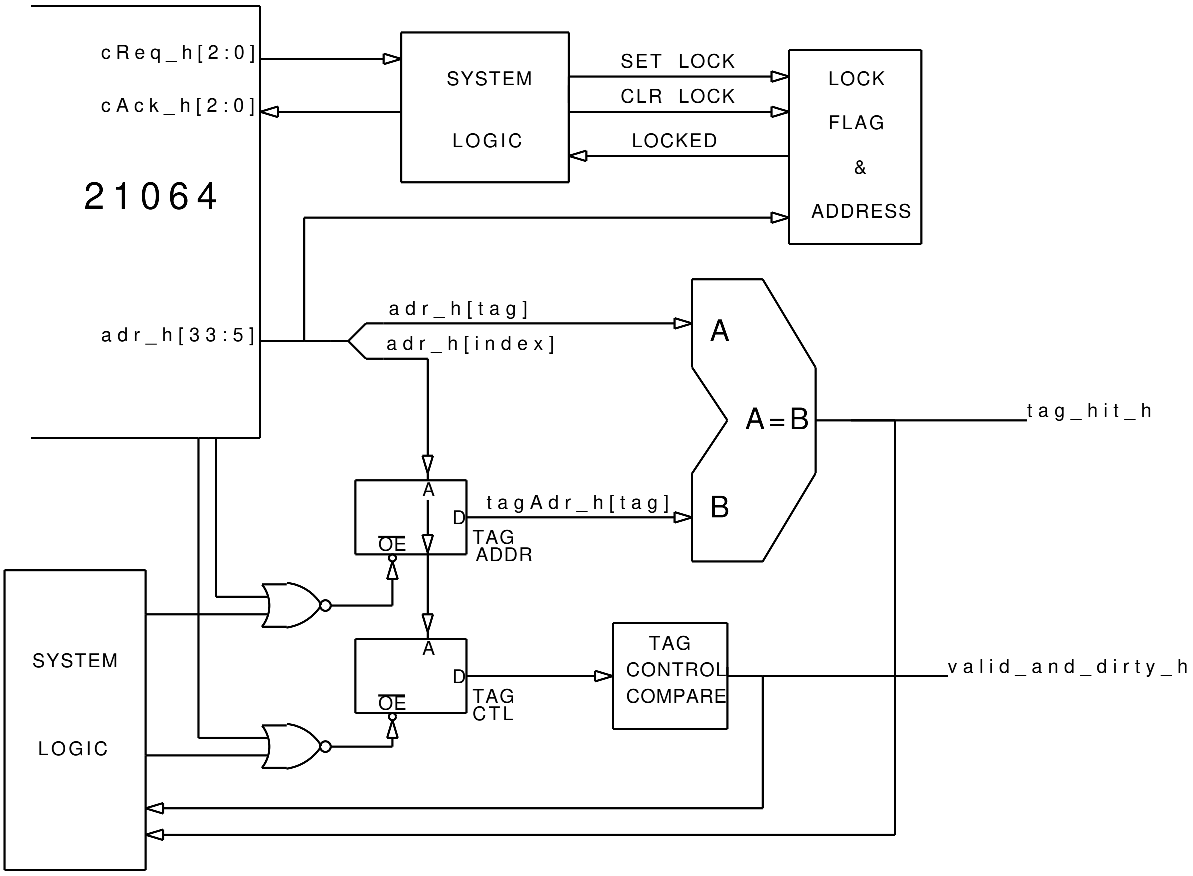
The 21064 provides the ability to perform locked memory accesses through the LDxL (Load_Locked) and STxC (Store_Conditional) cycle command pair. The LDxL command will force the 21064 to bypass the Bcache and request data directly from the external memory interface. The memory interface logic must set a special interlock flag as it returns the data, and may optionally keep other information about the transaction (such as the locked address).

The data requested for the LDxL access might be in the Bcache, since it has not been probed, so the external memory logic must do its own probe to determine where to obtain the information. In previous descriptions, the system logic only had to probe the tag control VALID and DIRTY RAMs to determine if a victim write was necessary. For the LDxL and STxC probe, the entire tag address must be compared, since the data that is being accessed might be in the Bcache.

Figure 26 shows a diagram of the probe and compare logic. On the initial request (the **cReq_h[2:0]** lines specify that the external LDxL must be performed) the system logic enables the tag RAMs and compares them to the tag field of the address for the 21064. If they compare and the block is valid, then the data requested is already in the Bcache. If the tag compare also shows that the block is dirty, then the *only* place the data resides is in the Bcache. There are two choices:

1. The data can be accessed from the Bcache.
2. The data can be written back to memory, then accessed from there.

Figure 26: Tag Address Compare Circuit



If the tag compares and the block is valid, but it is not dirty, then both the Bcache and the memory contain the data. It can be accessed from either place. If the tag fails or the block is *not* valid, then the data is only available from memory and must be accessed from there. In all the above cases, a flag must be set that signifies the location is locked.

Every design needs to provide a lock flag, but the amount of address information latched is completely up to the designer. On a uniprocessor system that does not expect much lock contention, simply having the lock flag with no address information might be enough. If any device accesses a memory location, the flag can be cleared, which will cause the subsequent store cycle to fail. On a multiprocessor system that expects real lock contention, lock address information can be saved so that different processors can lock different areas.

The STxC instruction is executed by the 21064 to clear the lock (and to find out if the code that was executed did so without contention). It is a write-type request where the processor bypasses the Bcache without a probe. If no other access has been made to the locked data, the STxC is treated similarly to a regular external memory write, though the Bcache must be probed by the system logic to determine where the most up-to-date data is located. The locked flag is also cleared.

If the Bcache probe finds that the data is both valid and dirty, the choices are similar to the read case:

1. The data can be written into the Bcache, using the **cWMask_h[7:0]** to determine which 4-byte segments should be modified. The STxC command will never validate more than a single 4-byte or 8-byte segment of data, and this can be used to optimize the cycle if desired.
2. The data can be written back to memory with a victim write, and modified there.

If the locked data location has been accessed between the LDxL and STxC commands, the external memory logic must return a special acknowledgment code that notifies the 21064 of this fact. In this case, no Bcache probe or actual external cycle needs to be performed.

7 Special Request Cycles

There are some external request cycles that might not actually perform any work, but must still provide the 21064 with an acknowledgment. The BARRIER, FETCH, and FETCH_M cycles are described in the preliminary data sheet, and will perform a system-specific function. When they are sensed by the external control logic, the system must minimally provide a **cAck_h** acknowledgment of OK.

8 DMA Access

There are situations where a device connected to an I/O bus needs direct access to the 21064 cache/memory subsystem. In the most general case the data could be in the Bcache, and that is the one discussed in this section. If a restriction can be made that eliminates the possibility of the target data being in the Bcache (that is, the DMA is always done to uncached space), then a simpler version of this discussion applies.

There are several ways that the external logic can perform a DMA access, the most straightforward of which is the use of the **holdReq_h** line. When a DMA device requires access to the 21064 cache/memory subsystem, it can notify the chip of that fact by asserting the **holdReq_h** signal. The 21064 replies to this request by asserting the **holdAck_h** signal. This signifies that the 21064 is no longer asserting the address, data, or Bcache control signals. The entire memory subsystem and Bcache are now under control of the external system logic.

The signal **holdAck_h** changes simultaneously with **sysClkOut1_h**. As such it should be sampled on an edge other than **sysClkOut1_h** if used as an input into state machines that run on **sysClkOut1_h**. This is similar to how **cReq_h[2:0]** must be used, as shown in Figure 14.

If it is assumed that the DMA target data (read or write) might be in the Bcache, the external logic must do a Bcache probe. This is similar to the probe necessary to determine if the data is in the Bcache when a LDxL or STxC is executed. The tag address and control RAMs should be compared to find out if the requested data is in the Bcache, and if it is dirty. The DMA logic can use the LDxL/STxC compare logic shown in Figure 26, or it can duplicate that logic for its own comparison.

The 21064 provides a third option for the tag address comparison, and this is the **tagEq_l** signal. When the chip is in **holdReq_h** mode, the **adr_h[33:5]** signals become inputs. The DMA device can drive its address on those lines and simultaneously enable the tag address RAMs. If the tag address compares with good parity, the signal **tagEq_l** will be asserted low. Consult the preliminary data sheet for more details about the use and timing of this feature.

For DMA read cycles where the probe shows that the data is valid in the Bcache, the choices are similar to what they were for the LDxL/STxC probe. If the data is valid but not dirty, it can be accessed from wherever it is most convenient. If the data is valid and dirty, it can be accessed directly from the Bcache or written back to memory and accessed there.

For a DMA write that hits in the Bcache, there are several choices:

1. The data can be written directly into the Bcache with the correct ECC or parity. In this case, the tag control should be made DIRTY, and the **dInvReq_h** signal should invalidate the cache line in the internal Dcache.
2. The data can be written back to memory with a victim write, and it can be modified there. The **dInvReq_h** signal should be asserted during the victim write or the DMA memory write to invalidate any stale Dcache data.

If the Bcache probe misses, or if the DMA access is defined to be only in the memory, then it is most sensibly accessed or modified there.

After the read or write cycle is complete, the **holdReq_h** signal can be de-asserted, which will cause the 21064 to de-assert the **holdAck_h** signal. The 21064 will then take control of the bus again, after a short delay.

There is one subtlety that should be mentioned here in regard to DMA access design. The 21064 might be in the middle of its own external (non-Bcache) access when it receives the **holdReq_h** request signal. If this happens, the chip might be waiting for data of its own, and has really only stalled the external cycle. As such, the data and cycle acknowledge signals are "live". The external logic must be careful not to assert the **dOE_l**, **dWSEL_h**, **dRack_h**, or **cAck_h** signals during its access cycle. Furthermore, there is a 2-CPU cycle delay between the time that the 21064 de-asserts the **holdAck_h** signal and when it re-enables its own address and data lines. This must be factored into the external logic for cycles that continue after the DMA stall.

In order to simplify the design, it is possible to filter the **holdReq_h** signal going to the 21064. If the external logic ensures that the **holdReq_h** signal only gets to the 21064 between cycles, then the problem of external cycles stalling in the middle is eliminated.

9 Backmapping the Internal 21064 Dcache

The 21064 provides the ability to keep a "backmap" of the internal Dcache tag address in external logic. In effect, the module adds enough extra information about the Dcache tag address to filter the invalidates that are sent to the 21064 Dcache. This can be used in multiprocessor systems or to filter DMA writes.

The processor outputs the signal **dMapWE_h** when it loads a block into the Dcache. This is meant to control an external memory array that takes the address from the appropriate **adr_h** lines and updates the external tag address memory location.

The external tag address does not have to contain the entire Dcache tag field, but rather needs only the difference between the Bcache and Dcache tag widths. If the Dcache is being kept as a subset of the Bcache, and if the Bcache is first probed, then the Dcache backmap is only responsible for knowing if a Bcache hit is also a Dcache hit. The preliminary data sheet describes the backmap in more detail.

10 I/O Interface

The input/output function of the 21064 is in some ways a subset of the memory function. I/O is normally not cached, so the probe will miss or not be performed at all for that memory quadrant. The access will go directly to the external interface bus as a READ_BLOCK or WRITE_BLOCK.

On a read cycle, the data will be returned as in the memory access already described, with the **dRack_h[2:0]** signals indicating that the data should be neither error-checked nor cached inside the chip. Since the return data is under complete control of the system interface logic, the Bcache will not be filled. On a write cycle, the steps are similar to a direct memory write cycle. The external logic can take the appropriate number of data words, then acknowledge the cycle.

The Alpha architecture provides an approach to I/O called a "mailbox". A description of the read or write is set up in memory. The description includes the full address, data, and mask information. A special mailbox register is then accessed in order to invoke the I/O transaction. This approach implies a smart I/O controller, and allows access to the full address range of the I/O bus.

If the mailbox option is not implemented, there are some techniques that can be employed when interfacing the 21064 to an I/O bus:

1. Address or data bits can be used to create byte masks and encode system level functions.
2. The 21064 address lines **adr_h** can be shifted right when accessing external buses that need the lower address bits. So, for example, **adr_h[20:5]** can translate to I/O address bits [15:0].
3. Reads and writes to I/O space can use the low bytes for all transactions, rather than pack the data into the appropriate field within the 32-byte block.
4. The **cWMask_h** field can normally be ignored for I/O writes.

11 Summary

The intent of this application note was to provide information so that a logic designer could understand the fundamental principles of creating a system with the 21064 processor chip. All of the chip's features were not covered, and it is suggested that the preliminary data sheet be consulted for more details about the 21064 and its use. Future application notes will cover different aspects of 21064 system design.