



SA-110 Microprocessor Instruction Timing:

An Application Note

Order Number: EC-R6M6A-TE

June 1997

This Application Note describes the timing of instructions and instruction sequences for the SA-110 microprocessor. It adds more detail to the information given in the *SA-110 Technical Reference* manual. From it, you should be able to calculate the time taken to execute any sequence of instructions.

Revision/Update Information: This is a new document.

Digital Equipment Corporation
Maynard, Massachusetts
<http://www.digital.com/semiconductor>

June 1997

While DIGITAL believes the information included in this publication is correct as of the date of publication, it is subject to change without notice.

Digital Equipment Corporation makes no representations that the use of its products in the manner described in this publication will not infringe on existing or future patent rights, nor do the descriptions contained in this publication imply the granting of licenses to make, use, or sell equipment or software in accordance with the description.

© Digital Equipment Corporation 1997. All rights reserved.

The following are trademarks of Digital Equipment Corporation: DIGITAL, Digital Semiconductor, and the DIGITAL Logo.

ARM is a registered trademark and StrongARM is a trademark of Advanced RISC Machines Ltd.

Digital Semiconductor is a Digital Equipment Corporation business.

Contents

1. Introduction	1
2. The StrongARM Core	2
2.1 The StrongARM Pipeline	2
2.1.1 Instruction Fetching	2
2.1.2 Multicycle Operations and Pipeline Stalls	2
2.1.3 Result Bypasses	3
2.2 Instruction Timings	3
2.2.1 Normal Instruction Processing	3
2.2.2 Branch Instructions	4
2.2.3 Data Processing Instructions	6
2.2.4 Load Word, Load Byte, and Load Halfword Instructions	8
2.2.5 Load Signed Halfword and Load Signed Byte Instructions	12
2.2.6 Store Instructions	13
2.2.7 Swap Instructions	13
2.2.8 Load and Store Multiple Processing	14
2.2.9 Multiplies	16
2.2.10 MSR Instructions	17
2.2.11 MRS Instructions	18
2.2.12 MCR Instructions	18
2.2.13 MRC Instructions	18
2.2.14 Instructions with the pc as a Target	18
2.2.15 Reading the pc in 26-bit Mode	22
2.2.16 Hardware Exception, Interrupt, and SWI Processing	22
3. The SA-110 Memory System	26
3.1 The Structure of the Memory System	26
3.2 Clocking	27
3.2.1 Clocking Domains	27
3.2.2 Phase of Generated MCLKs	27
3.2.3 Clock Switching	27
3.2.4 Bus Stalls	29
3.3 Instruction Memory System Timing	29
3.3.1 External Read Timing	29
3.3.2 Cache and TLB Hits	30
3.3.3 TLB Misses	30
3.3.4 Cache Misses, Uncacheable Instructions	31
3.3.5 Cache Misses, Cacheable Instructions	31
3.3.6 Instruction Side Coprocessor Access Timing	32
3.4 Data Memory System Timing	33
3.4.1 External Read Timing	33
3.4.2 Cache and TLB Hits	33

3.4.3 TLB Misses	33
3.4.4 Cache Misses, Uncacheable Data Reads	34
3.4.5 Cache Misses, Cacheable Data Reads	34
3.4.6 Reads Hitting the Write or Merge Buffer.....	35
3.4.7 Mergeable Writes Missing the Cache	36
3.4.8 Non-mergeable, Bufferable Writes Missing the Cache	36
3.4.9 Non-bufferable Writes Missing the Cache	36
3.4.10 Swaps Missing the Cache.....	37
3.4.11 Data Side Coprocessor Access Timing	38
3.4.12 Write Buffer Timing	38
3.5 The Bus Interface Unit.....	38
3.5.1 Bus Accesses and Cycles	38
3.5.2 Bus Contention Resolution	39

1. Introduction

The SA-110 can be viewed as a StrongARM (SA-1) core and a Harvard architecture memory system consisting of:

- A 16KB instruction cache
- A 16KB data cache
- Instruction and data memory management units, each containing a 32-entry translation look aside buffer
- An 8-entry write buffer
- A bus interface unit to access external memory
- A system control coprocessor controlling the memory system

This document describes the behavior of the StrongARM core and memory system in sufficient detail to allow the timing of any instruction sequence to be calculated. It assumes knowledge of the ARM architecture and of the StrongARM technical reference manual as prerequisites.

2. The StrongARM Core

2.1 The StrongARM Pipeline

The StrongARM pipeline has the stages shown in Table 2-1.

Table 2-1 Pipeline Stages

Stage	Action
0	Fetch stage – Fetch instruction from Icache or memory
1	Decode stage – Decode instruction, read input values from register file
2	Execute stage – Shifts and arithmetic (except multiplies)
3	Buffer stage – Data cache or memory access, multiplies, and system coprocessor access
4	Writeback – Write output values to register file

Most instructions normally spend a single cycle in each stage. The following sections describe the exceptional cases in which an instruction will spend more than one cycle in a single pipeline stage.

2.1.1 Instruction Fetching

The fetch stage is able to request an instruction from the Icache on every cycle. It will normally request the instruction sequentially following the instruction in the decode stage.

2.1.2 Multicycle Operations and Pipeline Stalls

Instructions may require more than one cycle in a pipeline stage for a number of reasons. Later sections describe precisely when this happens, but some of the reasons are:

- The instruction is waiting for the result of a previous instruction.
- The instruction inherently takes more than one cycle. An example of this would be a multiply instruction, which, depending on its arguments, may spend up to 3 cycles in the multiplier.
- The instruction is doing a memory access (or being fetched from memory) and this takes more than 1 cycle.

If, as a result of an instruction spending more than one cycle in a pipeline stage, the next pipeline stage becomes empty, then the processor will place a null instruction in this next pipeline stage. Once a null instruction is in the pipeline, it will spend one cycle in each remaining pipeline stage, unless the pipeline is stalled. A null instruction will always remain in the pipeline until it reaches the end of the pipeline.

An instruction will move between pipeline stages once it has completed the current pipeline stage and the next pipeline stage is available. If it completes its current pipeline stage before the next pipeline stage is available, it will stall in the current pipeline stage. This will normally stall all previous pipeline stages. If, however, a previous pipeline stage is executing a multicycle operation, then that stage will not stall until the multicycle operation completes.

Swaps, load and store multiples, and long multiplies are special cases. The processor fetches and decodes each of these as a single instruction, but the decode stage passes multiple instructions in the execute, buffer and writeback stages. Each of these generated instructions fills a slot in the pipeline, and will spend one or more cycles in each pipeline stage. The sections discussing these instructions give more details.

2.1.3 Result Bypasses

The StrongARM core contains a number of result bypasses. These normally allow the processor to use the results of one instruction in a following instruction as soon as it has been generated (and before it has been written back to the register file). In particular, almost every instruction can read its inputs from the bypasses as it enters the execute stage if these inputs are not yet in the register file in the decode stage. Later sections describe, in detail, the effects of these bypasses, and the few cases where these bypasses are not used.

2.2 Instruction Timings¹

2.2.1 Normal Instruction Processing

Table 2-2² shows how the processor processes most instructions if the pipeline does not stall, and if all instructions are immediately available from the Icache. The sections that follow describe the cases that do not follow this pattern.

¹All instruction timings in this section are described in terms of DCLK cycles. As described in the data sheet, and expanded upon in the next chapter, the length of DCLK cycles varies as the source of DCLK switches between the core clock (CCLK) and the bus clock (MCLK). This must be taken into account when calculating the elapsed time for any instruction sequence.

²All the timing diagrams in this chapter assume, unless otherwise stated, that all instructions are immediately available from the Icache and that all data accesses hit the Dcache. They also assume that, at the start of the sequence described, the pipeline is not stalled and the Dcache is not busy.

Table 2-2 Normal Instruction Processing

Cycle	Fetch Stage	Decode Stage	Execute Stage	Buffer Stage	Writeback Stage
1	Fetch from X	–	–	–	–
2	Fetch from X+4	Decode instruction at X	–	–	–
3	Fetch from X+8	Decode instruction at X+4	Execute instruction at X	–	–
4	–	Decode instruction at X+8	Execute instruction at X+4	Perform memory accesses, etc. for instruction at X	–
5	–	–	Execute instruction at X+8	Perform memory accesses, etc. for instruction at X+4	Write results of instruction at X to register file
6	–	–	–	Perform memory accesses, etc. for instruction at X+8	Write results of instruction at X+4 to register file
7	–	–	–	–	Write results of instruction at X+8 to register file

2.2.2 Branch Instructions

A branch instruction of the form B Y at address X behaves as shown in Table 2-3.

Table 2-3 Branch Instruction Processing

Cycle	Fetch Stage	Decode Stage	Execute Stage	Buffer Stage	Writeback Stage
1	Fetch from X	–	–	–	–
2	Fetch from X+4	Decode branch (B Y)	–	–	–
3	Fetch from Y	<i>Do nothing</i>	<i>No issue from decode (do nothing)</i>	–	–
4	–	Decode instruction at Y	<i>Do nothing</i>	<i>Do nothing</i>	–
5	–	–	Execute instruction at Y	<i>Do nothing</i>	<i>Do nothing</i>
6	–	–	–	Perform memory accesses, etc. for instruction at Y	<i>Do nothing</i>
7	–	–	–	–	Write results of instruction at Y to register file

Note that the processor calculates the destination in the decode stage and that a simple branch instruction does not use the execute, buffer, or writeback stages. If the fetch from X+4 misses the Icache or I-TLB, then the external fetch, cache line read, or fetch of the page table entry will complete before the processor fetches the instruction at Y. Note that if the fetch from X+4 misses both the Icache and the I-TLB, then the processor will only fetch the page table entry before fetching the instruction at Y. The branch instruction will stall before fetching from Y if, on cycle 3, the previous instruction is stalled in, or is still using, the execute stage.

A BL instruction of the form BL Y behaves exactly like the simultaneous execution of a B Y instruction and a SUB lr,pc,#4 instruction (using a pc value of X+8). See Table 2-4.

Table 2-4 Branch and Link Instruction Processing

Cycle	Fetch Stage	Decode Stage	Execute Stage	Buffer Stage	Writeback Stage
1	Fetch from X	–	–	–	–
2	Fetch from X+4	Decode branch (BL Y)	–	–	–
3	Fetch from Y	<i>Do nothing</i>	Execute SUB lr,pc,#4	–	–
4	–	Decode instruction at Y	<i>Do nothing</i>	Pass lr to writeback stage	–
5	–	–	Execute instruction at Y	<i>Do nothing</i>	Write lr to register file
6	–	–	–	Perform memory accesses, etc. for instruction at Y	<i>Do nothing</i>
7	–	–	–	–	Write results of instruction at Y written to register file

2.2.3 Data Processing Instructions

The processor normally processes data processing instructions at a rate of one instruction per cycle. The result of a data processing instruction is available, through bypasses, as soon as it has completed the execute stage of the pipeline. Any data processing instruction can read its inputs from the bypasses at the start of the execute stage.

The circumstances under which a data processing instruction will stall in the decode stage are:

- The previous instruction, which may be a null instruction, is still using or is stalled in the execute stage.
- The instruction requires a result generated by the buffer stage of a previous instruction (a memory access instruction, multiply, or system coprocessor access instruction), and that result is not yet available.

A data processing instruction will only stall in the execute stage if the buffer stage is still in use by the previous instruction.

All data processing instructions except those involving shifts by a register value require only 1 cycle in the execute and buffer stages. Data processing instructions involving shifts or rotations by a register value (a register shift instruction) require two cycles in the execute stage. When the processor executes a register shift instruction, the processor will not decode the following instruction until the second execute cycle. As an example of this, Table 2-5 shows the behavior of the following instruction sequence:

```
0      MOV r1, r2
4      MOV r3, r4, ROR r5
8      MOV r6, r7
```

The table assumes that all fetches hit the Icache and I-TLB.

Table 2-5 Register Shift Instruction Timing

Cycle	Fetch Stage	Decode Stage	Execute Stage	Buffer Stage	Writeback Stage
1	Fetch from X	–	–	–	–
2	Fetch from X+4	Decode MOV r1, r2. Load r2 from register file	–	–	–
3	Fetch from X+8	Decode MOVr3, r4, ROR r5. Load registers from register file	Execute MOV r1,r2	–	–
4	<i>Do nothing</i>	<i>Do nothing</i>	Execute MOVr3, r4, ROR r5 cycle 1	Buffer new value of r1	–
5	Fetch from X+12	Decode MOV r6, r7. Load r7 from register file	Execute MOVr3, r4, ROR r5 cycle 2	<i>Do nothing</i>	Write r1 to register file
6	–	–	Execute MOV r6, r7	Buffer new value of r3	<i>Do nothing</i>
7	–	–	–	Buffer new value of r6	Write r3 to register file
8	–	–	–	–	Write r6 to register file

2.2.4 Load Word, Load Byte, and Load Halfword Instructions

These load instructions, when they hit the Dcache (and D-TLBs), require one cycle in each pipeline stage. On a cache or TLB miss, the load instructions stay in the buffer stage until the requested data is available. They can read their inputs either from the register file during the decode stage, or from bypasses during the execute stage.

A load instruction may have 2 results:

- An updated base register. This is available through bypasses on completion of the execute stage.
- The value loaded. This is available through bypasses as soon as the load instruction leaves the buffer stage.

A load instruction will stall in the decode stage if:

- The execute stage is still in use by the previous instruction, or the previous instruction (which may be a NULL instruction) is stalled in the execute stage.
- The instruction requires a result generated by the buffer stage of a previous instruction (a memory access instruction, multiply, or system coprocessor access instruction), and that result is not yet available.

A load instruction will stall in the execute stage if the buffer stage is busy. Being busy means that either the previous instruction is still in the buffer stage, or that a previous instruction caused the Dcache to start a cache line fill that has not yet completed. Note that this is different from the conditions under which a data processing instruction stalls.

Table 2-6 shows the behavior of a load instruction followed by a data processing instruction that uses the result of the load instruction. The instruction sequence illustrated is:

```
0      LDR    r1, [r0,+4]!  
4      MOV    r2, r1
```

It assumes that the load hits the cache.

Table 2-6 Register Conflict Between a Load Instruction and a Following Data Processing Instruction

Cycle	Fetch Stage	Decode Stage	Execute Stage	Buffer Stage	Writeback Stage
1	Fetch from X	–	–	–	–
2	Fetch from X+4	Decode LDR r1, [r0,+4]	–	–	–
3	Fetch from X+8	Decode MOV r2, r1	Calculate r0+4	–	–
4	<i>Do nothing</i>	<i>Do nothing</i>	<i>Do nothing</i>	Read contents of r0+4 from cache	–
5	Fetch from X+12	Decode instruction at X+8	Execute MOV r2, r1 using bypass as input	<i>Do nothing</i>	Write new r0 and r1 to register file
6	–	–	–	Buffer new r2	<i>Do nothing</i>
7	–	–	–	–	Write new r2 to register file

For comparison, the following sequence, illustrated in Table 2-7, shows that the updated base register is available one cycle earlier. The instruction sequence considered is:

```

0      LDR   r1, [r0,+4]!
4      MOV   r2, r0

```

It assumes that the load hits the cache.

Table 2-7 Register Conflict Between a Load Instruction Base Register and a Following Data Processing Instruction

Cycle	Fetch Stage	Decode Stage	Execute Stage	Buffer Stage	Writeback Stage
1	Fetch from X	–	–	–	–
2	Fetch from X+4	Decode LDR r1, [r0,+4]	–	–	–
3	Fetch from X+8	Decode MOV r2, r0	Calculate r0+4	–	–
4	Fetch from X+12	Decode instruction at X+8	Execute MOV r2, r0 using bypass as input	Read contents of r0+4 from cache	–
5	–	–	–	Buffer new r2	Write new r0 and r1 to register file
6	–	–	–	–	Write new r2 to register file

Table 2-8 and Table 2-9 illustrate that a cache miss stalls a following load instruction for far longer than it stalls a following data processing instruction. Table 2-8 illustrates the timing of the sequence:

```

0      LDR   r1, [r0]
4      MOV   r3, r4

```

where the load misses the data cache.

Table 2-8 Cache Miss Followed by Data Processing Instruction

Cycle	Fetch Stage	Decode Stage	Execute Stage	Buffer Stage	Writeback Stage
1	Fetch from X	–	–	–	–
2	Fetch from X+4	Decode LDR r1, [r0]	–	–	–
3	Fetch from X+8	Decode MOV r3, r4	Calculate target address (r0)	–	–
4	Fetch from X+12	Decode instruction at X+8	Execute MOV r3, r4	Attempt to read data at r0 from cache and miss	–
5	<i>Do nothing</i>	<i>Do nothing</i>	<i>Do nothing</i>	Start cache fill	<i>Do nothing</i>
...
<i>n</i>	<i>Do nothing</i>	<i>Do nothing</i>	<i>Do nothing</i>	Data from r0 read from memory	<i>Do nothing</i>
<i>n</i> +1	Fetch from X+16	Decode instruction at X+12	Execute instruction at X+8	Buffer new r3 and continue cache fill	Write new r1 to register file
<i>n</i> +2	–	–	–	–	Write new r3 to register file

Table 2-9 shows the timing of the sequence:

```

0      LDR   r1, [r0]
4      LDR   r3, [r2]

```

where the first load misses the cache.

Table 2-9 Cache Miss Followed by Memory Access Instruction

Cycle	Fetch Stage	Decode Stage	Execute Stage	Buffer Stage	Writeback Stage
1	Fetch from X	–	–	–	–
2	Fetch from X+4	Decode LDR r1, [r0]	–	–	–
3	Fetch from X+8	Decode LDR r3, [r2]	Calculate target address (r0)	–	–
4	Fetch from X+12	Decode instruction at X+8	Calculate target address (r3)	Attempt to read data at r0 from cache and miss	–
5	<i>Do nothing</i>	<i>Do nothing</i>	<i>Do nothing</i>	Start cache fill	<i>Do nothing</i>
...
<i>n</i>	<i>Do nothing</i>	<i>Do nothing</i>	<i>Do nothing</i>	Data from r0 read from memory	<i>Do nothing</i>
<i>n+1</i>	<i>Do nothing</i>	<i>Do nothing</i>	<i>Do nothing</i>	Continue cache fill	Write new r1 to register file
...
<i>m</i>	<i>Do nothing</i>	<i>Do nothing</i>	<i>Do nothing</i>	Read final word of cache line	<i>Do nothing</i>
<i>m+1</i>	Fetch from X+16	Decode instruction at X+12	Execute instruction at X+8	Read data at r3 from cache	<i>Do nothing</i>
<i>m+2</i>	–	–	–	–	Write new r4 to register file

2.2.5 Load Signed Halfword and Load Signed Byte Instructions

These require at least 2 cycles in the buffer stage even if they hit the Dcache. If they miss the Dcache, they require no extra cycles beyond those required to fetch the data from the Dcache. Their timing is otherwise identical to load halfword and load byte instructions.

2.2.6 Store Instructions

Store instructions, when they hit the Dcache, normally require one cycle in each pipeline stage. They will require extra cycles in the buffer stage if:

- The address is neither cached nor bufferable
- The address is not cached and the write buffer is full
- The page table entry has to be read from external memory (it is not in the D-TLBs)

On a TLB miss, a store instruction stays in the buffer stage until the processor has read the required page table entries. Store instructions can read their inputs either from the register file during the decode stage, or from bypasses during the execute stage.

The only result a store instruction ever produces is an updated base register. This is available through bypasses when the instruction completes its execute stage.

A store instruction will stall in the decode stage if:

- The execute stage is still in use by the previous instruction, or the previous instruction (which may be a NULL instruction) is stalled in the execute stage.
- The instruction requires a result generated by the buffer stage of a previous instruction (a memory access instruction, multiply, or system coprocessor access instruction), and that result is not yet available.

A store instruction will stall in the execute stage if the buffer stage is busy.

2.2.7 Swap Instructions

A SWP or SWPB instruction behaves for timing purposes like a load instruction followed by a store instruction. Table 2-10 illustrates the timing of SWP r1,r2,[r0] on the assumption that the data is in the cache.

Table 2-10 Swap Instruction

Cycle	Fetch Stage	Decode Stage	Execute Stage	Buffer Stage	Writeback Stage
1	Fetch from X	–	–	–	–
2	Fetch from X+4	Decode SWP r1, r2, [r0] . Read r0 from register file.	–	–	–
3	<i>Do nothing</i>	Read r2 from register file	Calculate target address (r0)	–	–
4	Fetch from X+8	Decode instruction at X+4	Calculate target address (r0)	Read [r0] from cache	–
5	–	–	–	Write contents of r2 to [r0] in cache	Write new r1
6	–	–	–	–	<i>Do nothing</i>

Note that if the swap address is cacheable, but not cached, then the SWP-load will miss the cache, and the SWP-store will not enter the buffer stage until the cache line fill has completed.

2.2.8 Load and Store Multiple Processing

2.2.8.1 Simple Load and Store Multiples

Load and store multiples may be:

- Not in user mode
- Able to load or store at least 2 registers

Under these conditions, they take 1 pipe entry per register and have the same timing as the equivalent sequence of loads or stores would have. In particular, each component (single load or store) of a load or store multiple stalls under the same conditions as the equivalent load or store would.

Table 2-11 illustrates this for the case of LDMIA r1, {r2, r3, r4} where all the data is already in the cache.

Table 2-11 Simple Load Multiple Instruction

Cycle	Fetch Stage	Decode Stage	Execute Stage	Buffer Stage	Writeback Stage
1	Fetch from X	–	–	–	–
2	Fetch from X+4	Decode LDMIA r1, {r2, r3, r4} . Read r1 from register file	–	–	–
3	<i>Do nothing</i>	<i>Do nothing</i>	Calculate target address (r1)	–	–
4	<i>Do nothing</i>	<i>Do nothing</i>	Calculate target address (r1+4)	Read [r1] from cache	–
5	Fetch from X+8	Decode instruction at X+4	Calculate target address (r1+8)	Read [r1+4] from cache	Write new r2
6	–	–	–	Read [r1+8] from cache	Write new r3
7	–	–	–	–	Write new r4

If the instruction is to write back the base register, then the processor calculates its new value on the final execute cycle. The processor writes this new value to the register file on the final writeback cycle (the cycle after the final buffer cycle). As usual, most instructions can use bypasses to read the new value as soon as the execute cycle is complete.

Note that the buffer stage treats each component as a separate load or store instruction. This means that if a component causes a cache fill, then the next component cannot enter the buffer stage until that cache fill is complete.

2.2.8.2 Load and Store Multiples of 0 or 1 Registers

Load and store multiples of 0 or 1 registers always fill 2 pipe entries. The load or store, if any, starts on the first buffer cycle. The second execute cycle calculates the final base register value.

2.2.8.3 User Mode Load Multiple Instructions

The processor does not allow any further instructions to enter the execute stage until the final load has completed its writeback stage.

2.2.8.4 User Mode Store Multiple Instructions

After decoding a user mode store multiple instruction, the decode stage will stall for at least 2 cycles and until all previous instructions have completed their writeback stages.

2.2.9 Multiplies

The multiplier-accumulator is a separate unit running in parallel with the execute (E) and buffer (B) stages of the pipe. However, use of the multiplier blocks other instructions from using the execute and buffer stages of the pipe for one or more cycles. For calculating instruction timing, the multiplier portion occupying the E-stage and the accumulator in the B-stage of the pipeline can be viewed as follows:

- The E-stage (multiply array) takes 1–3 cycles depending on the value of Rs.
- The B-stage takes one cycle for a MUL or MLA and two cycles for a UMULL, SMULL, UMLAL or SMLAL; one cycle for a 32-bit result, and two cycles for a 64-bit result.

The multiplier supports signed early termination on Rs, and retires 11 bits on the first buffer stage cycle and 12 bits on the second buffer stage cycle. In other words, if bits 31 to 11 are all 0 or all 1, then the multiply will spend only one cycle in the execute stage. If bits 31 to 23 (but not 31 to 11) are all 0 or all 1, then the multiply will spend two cycles in the execute stage. If neither of these is true, it will spend three cycles in the execute stage.

A multiply can read its inputs through bypasses at the start of the execute stage. The result of a multiply becomes available through bypasses when the instruction leaves the buffer stage.

If there are two consecutive multiplies in the instruction stream, then the second multiply stalls in the decode stage, and does not enter the execute stage until the first multiply completes its buffer stage.

A multiply can enter the buffer stage as soon as the previous instruction leaves the buffer stage. Outstanding cache fills do not prevent a multiply from entering the buffer stage.

A long multiply takes one extra cycle. The multiplier outputs RdLo first, then RdHi in the last buffer stage cycle.

If a multiply sets the condition code, there will be at least a 3-cycle delay between the multiply entering the execute stage and the next instruction entering the execute stage. This is due to the need to feed back the condition codes from the B-stage for conditional execution, requiring an additional cycle and a NUL instruction bubble in the E-stage.

The timings of multiply and accumulate instructions are the same as those of multiply instructions.

2.2.10 MSR Instructions

Any MSR instruction that sets the CPSR mode bits (including the I and F bits) introduces 3 empty slots into the pipe. Table 2-12 illustrates the timing of MSR CPSR_c, r0.

Table 2-12 MSR CPSR_c Timing

Cycle	Fetch Stage	Decode Stage	Execute Stage	Buffer Stage	Writeback Stage
1	Fetch from X	–	–	–	–
2	Fetch from X+4	Decode MSR CPSR_c, r0	–	–	–
3	<i>Do nothing</i>	<i>Do nothing</i>	Execute MSR (cycle 1)	–	–
4	<i>Do nothing</i>	<i>Do nothing</i>	Set new mode	<i>Do nothing</i>	–
5	<i>Do nothing</i>	<i>Do nothing</i>	<i>Do nothing</i>	<i>Do nothing</i>	<i>Do nothing</i>
6	Fetch from X+8	Decode instruction at X+4	<i>Do nothing</i>	<i>Do nothing</i>	<i>Do nothing</i>
7	–	–	–	<i>Do nothing</i>	<i>Do nothing</i>
8	–	–	–	–	<i>Do nothing</i>

This sequence means that if the fetch of the next instruction hits the I-TLBs and Icache, permission faults on fetching this instruction are based on the old value of the CPSR. If it misses the I-TLBs or Icache, the processor will use the new value of the CPSR to check for permission faults.

Any MSR instruction that sets the CPSR condition code bits (but not the mode bits) introduces 2 empty slots into the pipe. Table 2-13 illustrates this for the instruction MSR CPSR_f, r0.

Table 2-13 MSR CPSR_f Timing

Cycle	Fetch Stage	Decode Stage	Execute Stage	Buffer Stage	Writeback Stage
1	Fetch from X	–	–	–	–
2	Fetch from X+4	Decode MSR CPSR_c, r0	–	–	–
3	<i>Do nothing</i>	Decode instruction at X+4	Execute MSR	–	–
4	<i>Do nothing</i>	<i>Do nothing</i>	Set new flag bits	<i>Do nothing</i>	–
5	Fetch from X+8	<i>Do nothing</i>	<i>Do nothing</i>	<i>Do nothing</i>	<i>Do nothing</i>
7	–	Decode instruction at X+8	Execute instruction at X+4	<i>Do nothing</i>	<i>Do nothing</i>
8	–	–	–	–	<i>Do nothing</i>

MSR instructions writing to an SPSR are single-cycle instructions that do not introduce any extra delays.

2.2.11 MRS Instructions

An MRS instruction has the same timing as a single-cycle data processing instruction. The CPSR or SPSR is read in the execute stage.

2.2.12 MCR Instructions

An MCR instruction cannot read from the register bypasses. As such, it has to wait for any preceding instruction that writes to its input register to complete its writeback stage before the MCR instruction can enter its execute stage. MCR instructions cannot enter the buffer stage while there is a cache fill in progress.

2.2.13 MRC Instructions

An MRC instruction has the same timing as a single-cycle load. An MRC instruction cannot enter the buffer stage while there is a cache fill in progress.

2.2.14 Instructions with the pc as a Target

Once the decode stage has identified that the current instruction has the pc as a target, it stops fetching or decoding further instructions until the new value of the pc is available (normally through bypasses). For most instruction types, the timing of the instruction that sets the pc is identical to the timing of an equivalent

instruction setting any other register. The fetch stage will, however, have already started to fetch the instruction that follows the instruction that writes to the pc. The processor starts to fetch the instruction at the new pc on the cycle on which the new pc becomes available.

The result of this is that if the sequence:

```
INSTR1      rx,...  ; rx is a target of this instruction
INSTR2      ...,rx  ; rx is an input to this instruction
```

starts executing INSTR2 on cycle n , then in the sequence:

```
INSTR1      pc,... ; Instruction with pc target
.
.
.
INSTR3 ...    ; Instruction at new pc.
```

INSTR3 will first be requested from the Icache on cycle n .

This means that if there are no other stalls or interlocks, a data processing instruction with the pc as its target will leave 2 empty slots in the pipe, and a load instruction will leave 3 empty slots in the pipe.

The sections that follow describe special cases that do not follow these rules.

2.2.14.1 *MOV pc,rx*

Special paths exist within the processor to ensure that instructions of the form *MOV pc,rx* only leave one empty slot in the pipe.

If rx contains Y, the following instruction at address X behaves as shown in Table 2-14:

```
MOV pc, rx
```

Table 2-14 MOV pc, rx Instruction Processing

Cycle	Fetch Stage	Decode Stage	Execute Stage	Buffer Stage	Writeback Stage
1	Fetch from X	–	–	–	–
2	Fetch from X+4	Decode instruction at X (MOV pc,rx)	–	–	–
3	Fetch from Y	<i>Do nothing</i>	<i>Do nothing</i>	–	–
4	–	Decode instruction at Y	<i>Do nothing</i>	<i>Do nothing</i>	–
5	–	–	–	<i>Do nothing</i>	<i>Do nothing</i>
6	–	–	–	–	<i>Do nothing</i>
7	–	–	–	–	Write results of instruction at Y written to register file

The MOV instruction will stall in the decode stage, before starting the new fetch, if rx is the destination of a previous instruction that is not yet in its writeback stage in cycle 2. The MOV pc, rx instruction **cannot** use bypasses to read rx. It will also stall without starting the fetch if, on cycle 3, the previous instruction is still using, or is stalled in, the execute stage.

If there are no such stalls and no register conflicts, a MOV rx, pc instruction will leave 1 empty slot in the pipe. A register conflict (even with a data processing instruction) can leave up to 3 empty slots in the pipe.

Note: A surprising consequence of not using bypasses is that the instruction sequence:

```
MOV    r3, r2    ; Or any other data processing instruction writing r3.
ADD    pc, r3, #0
```

is, on StrongARM, one cycle faster than the sequence:

```
MOV    r3, r2
MOV    pc, r3
```

Only instructions of the form MOV pc, rx use the special path described in this section. In particular, MOVS pc, rx instructions and MOV pc,<shifter_operand> instructions in which <shifter_operand> is more complex than an unmodified register do not use this path.

2.2.14.2 Restore CPSR Data Processing Instructions (MOVS pc,rx, etc.)

Any restore CPSR instruction will introduce at least 3 empty slots into the pipe between decoding the restore CPSR instruction and decoding the instruction at the return address. Table 2-15 shows, as an example, the timing for MOVS pc,r14 assuming that r14 contains Y.

Table 2-15 MOVS pc, r14 Instruction Processing

Cycle	Fetch Stage	Decode Stage	Execute Stage	Buffer Stage	Writeback Stage
1	Fetch from X	–	–	–	–
2	Fetch from X+4	Decode instruction at X (MOVS pc,r14)	–	–	–
3	<i>Do nothing</i>	<i>Do nothing</i>	Execute MOVS pc,r14	–	–
4	<i>Do nothing</i>	<i>Do nothing</i>	<i>Do nothing</i>	Buffer MOVS pc,r14 (update CPSR)	–
5	Fetch from Y	<i>Do nothing</i>	<i>Do nothing</i>	<i>Do nothing</i>	<i>Do nothing</i>
6	–	Decode instruction at Y	<i>Do nothing</i>	<i>Do nothing</i>	<i>Do nothing</i>
7	–	–	Execute instruction at Y	<i>Do nothing</i>	<i>Do nothing</i>
8	–	–	–	Buffer instruction at Y	<i>Do nothing</i>
9	–	–	–	–	Write results of instruction at Y to register file

2.2.14.3 Load and Store Multiple Instructions Writing to the pc

These instructions behave as defined in Section 2.2.8. The pc can be updated as the base register (data loads from the instruction stream) or as the last register in the register list:

LDMIA pc!, {r4, r5, r6}

If the pc is the base register, it will start simultaneously with the final load:

LDMIA r0, {r4, r5, r6,pc}

If the pc is the last load target, the fetch of the branch destination will start in the cycle following completion of the final load.

2.2.15 Reading the pc in 26-Bit Mode

Consider the case where, in 26-bit mode, a data processing instruction (not a multiply) sets the condition code and the following instruction is:

- A BL instruction.
- Any instruction, except swaps, with bits 0 to 3 of the instruction set to 15 and bits 25 to 27 set to 0. This includes data processing instructions for which register 15 is the rm argument, but also includes certain other instructions such as LDRH r0, [r1, #15].
- A swap or swap byte using register 15 for Rd (the destination register).
- A store instruction for which register 15 is the register to be stored.

The second instruction will then stall in the decode stage for one cycle after the first instruction leaves the execute stage.

2.2.16 Hardware Exception, Interrupt, and SWI Processing

2.2.16.1 Undefined Instruction Exception

Table 2-16 shows the timing of an undefined instruction.

Table 2-16 Undefined Instruction Processing

Cycle	Fetch Stage	Decode Stage	Execute Stage	Buffer Stage	Writeback Stage
1	Fetch from X	–	–	–	–
2	Fetch from X+4	Decode instruction at X (undefined)	–	–	–
3	<i>Do nothing</i>	Second decode cycle	<i>Do nothing</i>	–	–
4	Fetch from X+8	<i>Do nothing</i>	Calculate new r14	<i>Do nothing</i>	–
5	Fetch from X+12	<i>Do nothing</i>	<i>Do nothing</i>	Buffer new r14	<i>Do nothing</i>
6	Fetch from 4	<i>Do nothing</i>	Set new CPSR and SPSR (at start of cycle)	<i>Do nothing</i>	Write back new r14
7	–	Decode instruction at 4	<i>Do nothing</i>	<i>Do nothing</i>	<i>Do nothing</i>
8	–	–	Execute instruction at 4	<i>Do nothing</i>	<i>Do nothing</i>
9	–	–	–	Buffer instruction at 4	<i>Do nothing</i>
10	–	–	–	–	Writeback results of instruction at 4

If the execute or buffer stage stalls at the end of cycle 2, then the processor will put one less bubble in the pipe (i.e., cycle 3 executes in the decode stage on the cycle following cycle 2 even if the pipe is stalled).

All other stalls will stall the pipe as usual.

2.2.16.2 Software Interrupt

The timing of a software interrupt is shown in Table 2-17.

Table 2-17 SWI Processing

Cycle	Fetch Stage	Decode Stage	Execute Stage	Buffer Stage	Writeback Stage
1	Fetch from X	–	–	–	–
2	Fetch from X+4	Decode instruction at X (SWI)	–	–	–
3	Fetch from X+8	<i>Do nothing</i>	Calculate new r14	<i>Do nothing</i>	–
4	Fetch from X+12	<i>Do nothing</i>	<i>Do nothing</i>	Buffer new r14	<i>Do nothing</i>
5	Fetch from 8	<i>Do nothing</i>	Set new CPSR and SPSR (at start of cycle)	<i>Do nothing</i>	Write back new r14
6	–	Decode instruction at 8	<i>Do nothing</i>	<i>Do nothing</i>	<i>Do nothing</i>
7	–	–	Execute instruction at 8	<i>Do nothing</i>	<i>Do nothing</i>
8	–	–	–	Buffer instruction at 8	<i>Do nothing</i>
9	–	–	–	–	Writeback results of instruction at 8

2.2.16.3 Prefetch Abort

Once the fetch stage has seen an abort, it will stop fetching instructions. An explicit change to the pc, caused either by taking an exception or by executing a branch instruction or an instruction that writes to the pc, will restart instruction fetch.

Otherwise, the timing of a prefetch abort is identical to the timing of an SWI, i.e., it puts 3 bubbles in the pipeline.

2.2.16.4 Data Abort

If a data abort occurs, the processor fetches the instruction at address 0x10 and changes the CPSR on the cycle following the data abort. Furthermore, a data abort cancels all stalls and prevents any results from instructions in the pipeline from being written back.

2.2.16.5 Interrupts

The timing for IRQs and FIQs is identical. The timing is controlled as follows:

- There is a delay of two core cycles between the interrupt pin changing state and the processor core seeing the change. This delay is unaffected by any pipe stalls.
- Once the interrupt reaches the core, it checks whether interrupts are enabled. If they are, then the processor stalls the fetch stage until the instruction pipe is empty, interrupts are disabled, or the core sees the interrupt being cleared.
- On each cycle following this, the core checks the state of the pipe. Once there has been a 2-cycle delay after the last instruction leaves the buffer stage, then on the following cycle the processor updates the CPSR and fetches the instruction in the interrupt vector. Note that for the purposes of checking, the pipe instructions that fail their condition code check are ignored.

Typical timing of an interrupt is show in Table 2-18.

Table 2-18 Interrupt Timing

Cycle	Action
1	nIRQ asserts
2	
3	
4	Interrupt reaches core
5	Fetch stopped, last instruction in decode
6	
7	
8	Last instruction in writeback
9	
10	
	Instruction at 0x18 fetched

If an instruction enables interrupts, then this does not take effect for 2 cycles after the CPSR is modified. If an instruction disables interrupts, then this takes effect immediately. As such, the processor only acts on interrupts if interrupts have been enabled for at least 2 consecutive cycles.

3. The SA-110 Memory System

3.1 The Structure of the Memory System

The SA-110 has independent internal instruction and data memory systems. Both of them access a common external memory system through a common bus interface unit (BIU). The system coprocessor controls the two memory systems. This coprocessor is implemented as a set of registers in the data memory system. The instruction memory system contains duplicates of those registers that affect its behavior. These duplicates are write-only registers.

The StrongARM core fetches instructions from the instruction memory system. It contains the instruction cache (Icache) and a translation lookaside buffer (TLB) containing 32 entries. The Icache is 16KB read-only cache consisting of 512 lines of 32 bytes. It is organized as a 32-way associative cache and is virtually addressed.

The StrongARM core passes all data accesses resulting from memory access instructions to the data memory system. It contains the data cache (Dcache), a 32-entry TLB, a write merge buffer and an 8-entry write buffer. The Dcache is a 16KB write-back cache consisting of 512 lines of 32 bytes. It is organized as a 32-way set associative cache and is virtually addressed. The write merge buffer is 16 bytes wide, and will merge consecutive mergeable writes to the same half cache line. It is only flushed into the write buffer when:

- The write buffer is explicitly flushed
- A non-mergeable write occurs
- A write occurs to a different half cache line

The write buffer contains 8 entries of 16 bytes each. All writes, even non-bufferable writes, go through the write buffer. The processor performs a non-bufferable write by placing the data in the write buffer and then flushing the write buffer.

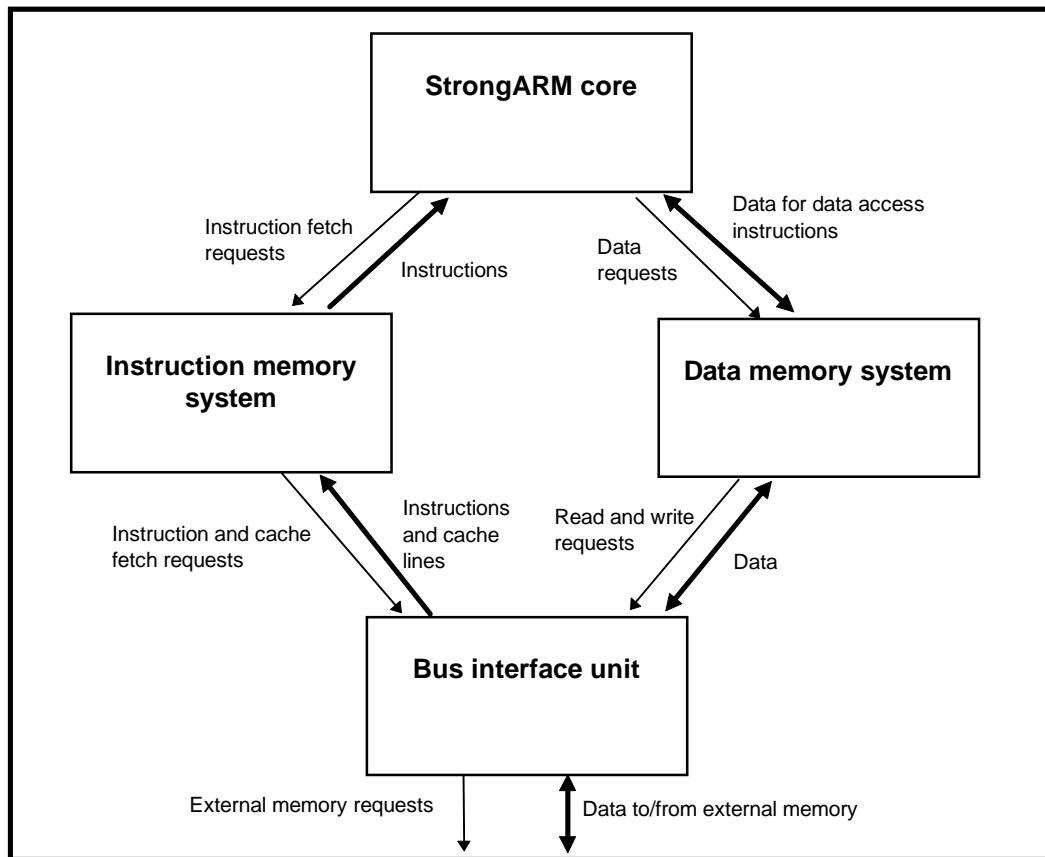
Although the instruction and data TLBs are separate, they share a single page table base and hence a single set of page tables.

The BIU controls the order of external bus accesses when there is more than one access outstanding. It also performs the actual external bus cycles.

The memory system also controls clock switching. The StrongARM core will run at the core clock speed when there is no external access (if clock switching is enabled). It switches to running at the bus clock speed when an external access is in progress.

Figure 3-1 shows the structure of and the command and data flows in the SA-110's memory system.

Figure 3-1 SA-110 Memory System



3.2 Clocking

3.2.1 Clocking Domains

The SA-110 contains 2 clocking domains. The DCLK domain includes the StrongARM core, the internal instruction memory system, and all of the data memory system except for the output stages of the write buffer. The output stages of the write buffer, the BIU, and the external interfaces are clocked at MCLK. CCLK is defined as the output of the internal phase lock loop (PLL), the frequency dependent on both the reference oscillator and state of the CCCFG[3:0] pins. The DCLK domain is clocked by CCLK or MCLK depending on the switching logic state.

3.2.2 Phase of Generated MCLKs

If the processor is configured to generate a synchronous MCLK, then MCLK rising edges will always correspond to CCLK rising edges.

3.2.3 Clock Switching

The data sheet explains that the clock source for the DCLK domain switches between the internal, fast, CCLK, and the slower MCLK. When switching the

DCLK source from CCLK to MCLK, the processor extends the DCLK low stage of the cycle to the next rising edge of MCLK after the first MCLK falling edge. Figure 3-2 and Figure 3-3 show two examples of this.

Figure 3-2 Switching the DCLK Source to MCLK (Example 1)

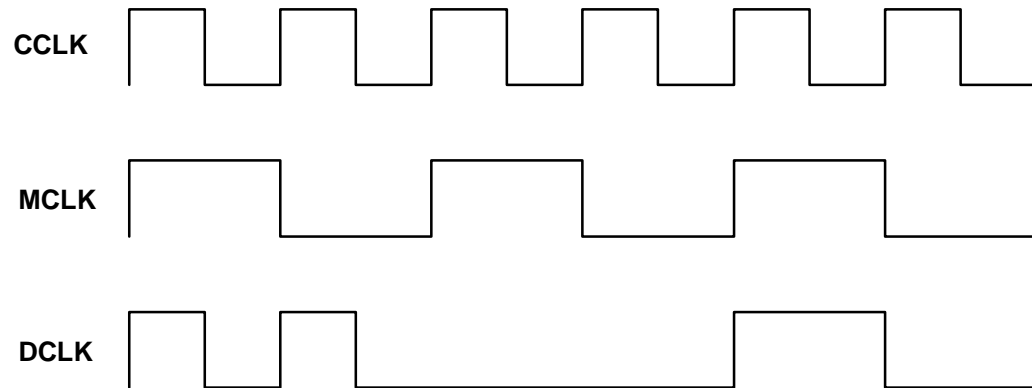
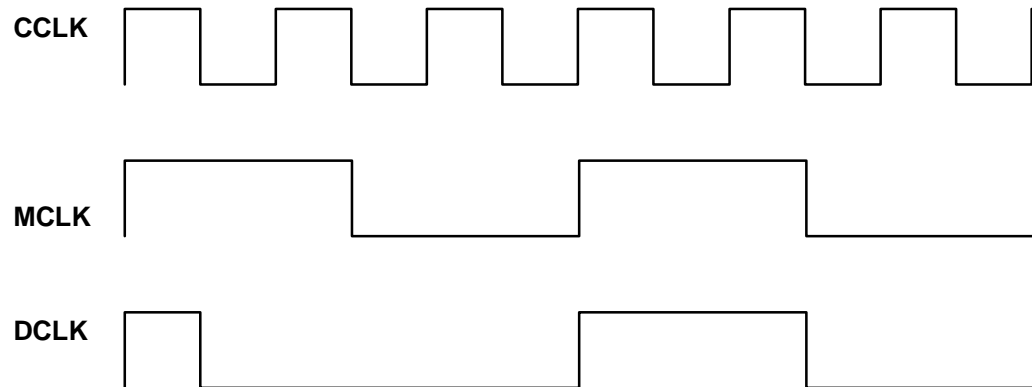


Figure 3-3 Switching the DCLK Source to MCLK (Example 2)



Switching from sourcing DCLK from MCLK to sourcing from CCLK takes between 1.5 and 2.5 CCLK cycles. The switch always happens in the DCLK low phase. The phase will start with the MCLK falling edge. It will end on the CCLK rising edge following 2 CCLK falling edges during the phase, in addition to any CCLK falling edge that happens simultaneously with the MCLK falling edge. The minimum time required between the MCLK falling edge and the first CCLK falling edge (when working with asynchronous clocks) is not defined, but will be less than half a CCLK cycle. Figure 3-4 and Figure 3-5 show two examples of this.

Figure 3-4 Switching the DCLK Source to CCLK (Example 1)

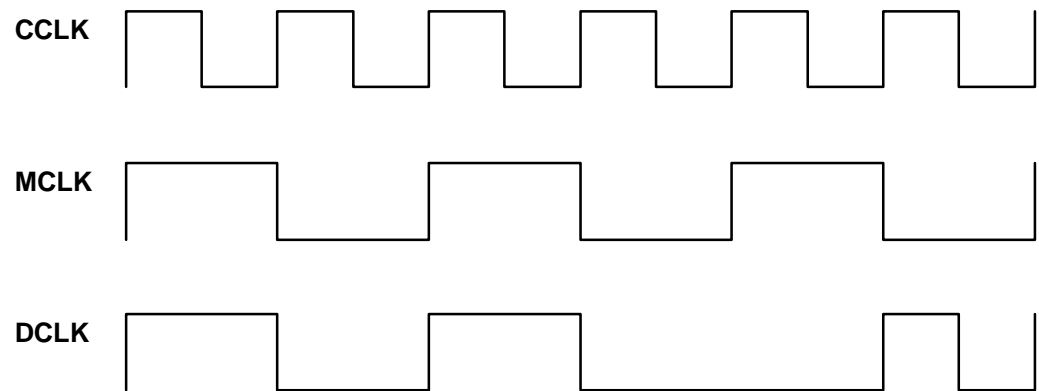
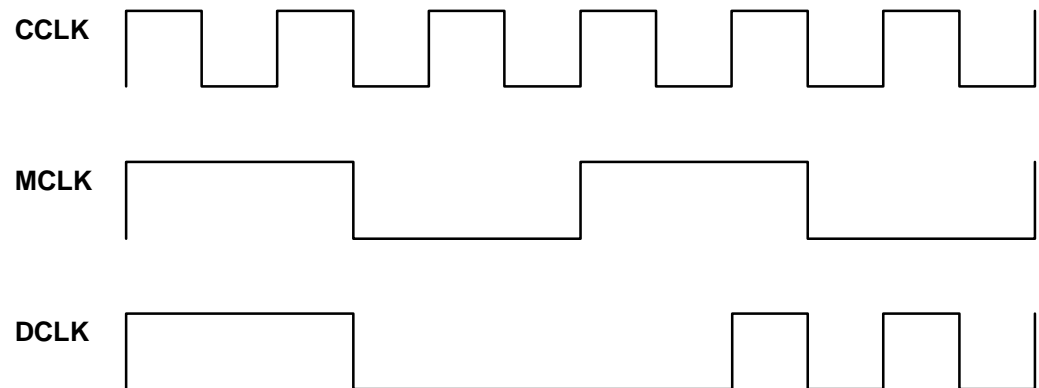


Figure 3-5 Switching the DCLK Source CCLK (Example 2)



3.2.4 Bus Stalls

If nWAIT is low on any MCLK rising edge, then nothing within the processor clocked off MCLK will be clocked until the next MCLK rising edge. In particular, if DCLK is being sourced from MCLK, the entire processor will do nothing for that cycle. If, however, DCLK is being sourced from CCLK, or is in the process of switching to being sourced from CCLK, then the DCLK clocking domain will continue to run as normal.

3.3 Instruction Memory System Timing

3.3.1 External Read Timing

The timing of any single word read requested by the instruction memory system is identical. Table 3-1 shows the timing of such a read assuming that there are no other outstanding requests. If there are other requests in progress, or a higher

priority request reaches the BIU simultaneously with this request, then extra cycles will be inserted between cycles 2 and 3 until the BIU can accept the request.

Bus stalls will extend DCLK cycles as described in Section 3.2.4.

Table 3-1 Single Word Reads by the Instruction Memory System

DCLK Cycle ³	Action
1	Instruction memory system discovers the need for a single word read (page table entry read or instruction fetch).
2 DCLK high	
2 DCLK low	Request sent to BIU. DCLK source switches to MCLK (if not already MCLK).
3 DCLK high	BIU receives request; if APE is high, the BIU writes the signals with address timing (A, MAS, nRW, CLF).
3 DCLK low	BIU sets SEQ and clears nMREQ; if APE is low, the BIU writes the signals with address timing.
4 DCLK high	
4 DCLK low	BUI clears SEQ and sets nMREQ.
5 DCLK high	Requester told result of read (success or abort).
5 DCLK low	Data read from D pins and available to requester. If no data side reads are outstanding, and clock switching is enabled, DCLK source switches to CCLK.

3.3.2 Cache and TLB Hits

A request for an instruction that hits the cache and TLB takes one cycle. The instruction is available to the core's decode stage in the cycle following the cycle in which the request for the instruction is made by the core's fetch stage.

3.3.3 TLB Misses

If a request for an instruction misses the TLB, then the instruction memory system reads the appropriate first-level page table entry from external memory. If this is a pointer to a second-level page table, then it reads the appropriate second-level page table. If the page table entries are valid, then it will write the new TLB entry to the TLB on the final cycle of the first or second read. If no stalls occur, this will be the 5th or 10th cycle. On the following cycle, if the fetch stage of the pipeline is still requesting the same instruction, the instruction memory system will restart the request for the instruction. This will now hit the TLB, and may or may not hit the cache.

³ In this document, DCLK cycles are defined to start on the DCLK rising edge. This corresponds to the behavior of the StrongARM core, which moves instructions along the pipeline on the rising edge. It is, however, the converse of the convention used in the data sheet, which shows external cycles starting on the MCLK falling edge.

If an instruction fetch misses both the TLB and the cache, then the instruction memory system will handle the TLB miss first (it does not know how to resolve the cache miss until it has a TLB entry for the instruction).

If the page table entries are not valid, then an abort is generated on the final cycle, i.e., for a first-level page table, this is the only cycle, and for a second-level page table, this is the second cycle.

3.3.4 Cache Misses, Uncacheable Instructions

If an instruction fetch misses the cache, and the instruction is not cacheable (or the Icache is disabled), then the instruction memory system will read it from external memory. It will be available to the decode stage on the cycle following the completion of the read. If no pipeline stalls occur due to the external read, this will be the 5th cycle following the request, thus placing 4 bubbles in the execution pipeline.

3.3.5 Cache Misses, Cacheable Instructions

If a cacheable instruction fetch misses the cache, then the instruction memory system reads the appropriate cache line. Once this has completed, if the fetch stage is still requesting the same instruction, the instruction memory will restart the request for the instruction. This will normally succeed in a single cycle, but may fail if an instruction in the execution pipeline has modified something that affects protection during the cache line fill. It is also possible for an instruction in the pipeline to have flushed the TLB during the fill, in which case a new page table read will be needed.

Table 3-2 shows the timing of a cache line fill. If there are other requests in progress, or a higher priority request reaches the BIU simultaneously with this request, then extra cycles will be inserted between cycles 2 and 3 until the BIU can accept the request.

Table 3-2 Cache Line Fill by the Instruction Memory System

DCLK Cycle	Action
1	Cacheable instruction fetch misses cache.
2 DCLK high	
2 DCLK low	Request sent to BIU. DCLK source switches to MCLK (if not already MCLK).
3 DCLK high	BIU receives request; if APE is high, the BIU writes the signals with address timing (A, MAS, nRW, CLF).
3 DCLK low	BIU sets SEQ and clears nMREQ; if APE is low, the BIU writes the signals with address timing.
4 DCLK high	
4 DCLK low	
5 DCLK high	Address on bus changed to second word if APE high.
5 DCLK low	First word read from D pins and placed in cache. Address on bus changed to second word if APE low.
6 DCLK high	Address on bus changed to second word if APE high.
6 DCLK low	Second word read from D pins and placed in cache. Address on bus changed to third word if APE low.
...	...
11 DCLK low	BUI clears SEQ and sets nMREQ. Seventh word read from D pins and placed in cache. Address on bus changed to eighth word if APE low.
12 DCLK high	
12 DCLK low	Eighth word of cache line read and placed in cache. Cache line marked valid. If no further reads are outstanding, and clock switching is enabled, DCLK source switches to CCLK.
13 DCLK high	Instruction fetch retried.
13 DCLK low	Instruction fetch hits cache.
14 DCLK high	Instruction decode starts.

3.3.6 Instruction Side Coprocessor Access Timing

All instruction side coprocessor accesses happen in the cycle following that in which the MCR instruction enters the buffer stage. All are single cycle accesses. If a cache or TLB flush happens on the same cycle as a cache or TLB fill completing, then the entry that has just been read will be invalidated.

3.4 Data Memory System Timing

3.4.1 External Read Timing

The timing of any single word read (or partial word read) requested by the data memory system is identical. Table 3-3 shows the timing of such a read assuming that there are no other outstanding requests. If there are other requests in progress, or a higher priority request reaches the BIU simultaneously with this request, then extra cycles will be inserted between cycles 3 and 4 until the BIU can accept the request.

Bus stalls will extend DCLK cycles as described in Section 3.2.4.

Table 3-3 Single Word Reads by the Data Memory System

DCLK Cycle	Action
1	Data memory system discovers the need for a single word read (page table entry read or data read).
2 DCLK high	
2 DCLK low	
3 DCLK high	
3 DCLK low	Request sent to BIU. DCLK source switches to MCLK (if not already MCLK).
4 DCLK high	BIU receives request; if APE is high, the BIU writes the signals with address timing (A, MAS, nRW, CLF).
4 DCLK low	BIU sets SEQ and clears nMREQ; if APE is low, the BIU writes the signals with address timing.
5 DCLK high	
5 DCLK low	BUI clears SEQ and sets nMREQ.
6 DCLK high	Requester told result of read (success or abort).
6 DCLK low	Data read from D pins and available to requester. If no data side reads are outstanding, and clock switching is enabled, DCLK source switches to CCLK.

3.4.2 Cache and TLB Hits

A read or write that hits the cache and TLB takes one cycle.

3.4.3 TLB Misses

If a read or write misses the TLB, then the data memory system reads the appropriate first-level page table from external memory. If this is a pointer to a second-level page table, then it reads the appropriate second-level page table. If the page table entries are valid, the data memory system will write the new TLB entry to the TLB on final cycle of the first or second read. If no stalls occur, this

will be the 5th or 10th cycle. On the following cycle, the data memory system will restart the read or write. This will now hit the TLB, and may or may not hit the cache.

If a read misses both the TLB and the cache, then the TLB miss will be handled first (it does not know how to resolve the cache miss until it has a TLB entry for the data).

If the page table entries are not valid, then the translation fault will be signalled on the final cycle of the read of the page table entry.

3.4.4 Cache Misses, Uncacheable Data Reads

If a data read misses the cache and the data is not cacheable (or the Dcache is disabled), then it will be read from external memory. The instruction (or instruction component for load multiples and swaps) requesting the data will enter the writeback stage, and the data will be available through the bypasses on the cycle following the completion of the read.

3.4.5 Cache Misses, Cacheable Data Reads

If a cacheable read misses, then the data memory system reads the appropriate cache line. The instruction, or instruction component requesting the read, will enter the writeback stage on the cycle following that in which the actual data requested is read from the cache. However, no other instruction can use the data memory system until the cache line fill completes.

Note

If the bus is running in enhanced mode, the actual data requested will always be in the first word read. If the bus is running in standard mode, then the data memory system will read the cache line in order starting from its low address. As such, cache misses to other than the first word of a cache line will take longer to make the data available to other instructions in standard mode than in enhanced mode. However, whichever bus mode is used, the memory system remains in use and unavailable for further instructions, for the same number of cycles.

Table 3-4 shows the timing of a cache line fill if the cache line being overwritten is empty, or there have been no writes to the cache line. If there are other requests in progress, or a higher priority request reaches the BIU simultaneously with this request, then extra cycles will be inserted between cycles 3 and 4 until the BIU can accept the request.

If the cache line being overwritten contains modified data, then this has to be written to the write buffer before the new data can be written to the cache. If the write buffer contains at least 3 free entries, then no additional cycles are required for this. If both halves of the cache line are dirty, the first half of the cache line is written to the write buffer in cycles 2 and 3, and the second half cache line is written to the write buffer in cycles 4 and 5. If only one half is dirty, then this is written to the write buffer in cycles 2 and 3.

If 3 entries are not available, then extra cycles will be inserted between cycles 1 and 2 until they are available.

Table 3-4 Cache Line Fill by the Data Memory System

DCLK Cycle	Action
1	Cacheable data read misses cache.
2 DCLK high	
2 DCLK low	
3 DCLK high	
3 DCLK low	Request sent to BIU. DCLK source switches to MCLK (if not already MCLK).
4 DCLK high	BIU receives request; if APE is high, the BIU writes the signals with address timing (A, MAS, nRW, CLF).
4 DCLK low	BIU sets SEQ and clears nMREQ; if APE is low, the BIU writes the signals with address timing.
5 DCLK high	
5 DCLK low	
6 DCLK high	Address on bus changed to second word if APE high.
6 DCLK low	First word read from D pins and placed in cache. Address on bus changed to second word if APE low.
7 DCLK low	Address on bus changed to second word if APE high.
7 DCLK high	Second word read from D pins and placed in cache. Address on bus changed to third word if APE low.
...	...
12 DCLK low	BUI clears SEQ and sets nMREQ. Seventh word read from D pins and placed in cache. Address on bus changed to eighth word if APE low.
13 DCLK high	
13 DCLK low	Eighth word of cache line read and placed in cache. Cache line marked valid. If no reads or unbuffered writes are outstanding, and clock switching is enabled, DCLK source switches to CCLK.
14 DCLK high	Data memory system ready for other instructions. BIU ready for other reads or writes.

3.4.6 Reads Hitting the Write or Merge Buffer

If the StrongARM core requests a read of an address in the write buffer, then the data memory system will write the entry containing this address and all entries preceding it in the write buffer using high-priority writes. This causes the writes

to happen before the data memory system reads the requested data, or any other data, from the bus. See Section 3.5.2.

If a read is requested of an address in the merge buffer, then the merge buffer is flushed into the write buffer, and all entries in the write buffer will be written using high-priority writes. If the write buffer is full, the instruction will stall until at least one entry is available.

3.4.7 Mergeable Writes Missing the Cache

A write is mergeable with previous writes if:

- It is to the same half cache line as the contents of the merge buffer
- It is cacheable and bufferable
- Either the processor is running in enhanced bus mode, or the write is the second or subsequent write of a store multiple

A mergeable write is written to the merge buffer, and completes in one cycle.

3.4.8 Non-Mergeable, Bufferable Writes Missing the Cache

All other bufferable writes will force the merge buffer into the write buffer (if the merge buffer is in use), and then write to the merge buffer. This will complete in one cycle unless the write buffer is full, in which case it will stall until there is at least one free entry in the write buffer.

3.4.9 Non-Bufferable Writes Missing the Cache

A non-bufferable write will stall until there is at least one free entry in the write buffer. It will then write its data to the write buffer and stall again until the external write of this data has been completed. Table 3-5 shows the timing of a non-bufferable write missing the cache if, at the start of the write, the write buffer is empty and no other reads or writes are in progress. The write will skip cycle 4 if, at the start of cycle 3, DCLK is sourced from CCLK, and the relative phases of CCLK and MCLK are such that MCLK drops at the start of, or during, the DCLK high phase of cycle 3. The reason for this is that the timing of cycles 3 and 4 is controlled by the following requirements:

- There must be two falling MCLK edges between data being written to the write buffer and it being passed to the BIU (between the rising edges of DCLK cycles 3 and 5).
- Clock switching requires that there must be one falling MCLK edge between the start and end of the DCLK low phase of cycle 3.

As such, cycle 4 is required as long as there is no rising MCLK edge during the DCLK high phase of cycle 3.

The effect of this is that the time from when the data enters the write buffer to when it enters the BIU is always greater than 1.5 and no more than 2.5 MCLK cycles.

If the write and merge buffers are full, the write stalls after cycle 1 until the merge buffer is free. If the write buffer is full, it stalls after cycle 2 until there is a free entry. If there are other entries in the write buffer, or there is some other I/O in

progress, it stalls after cycle 4 until all previous entries in the write buffer have been written and the BIU is available.

Table 3-5 Non-Bufferable Write Timing

DCLK Cycle	Action
1	Write misses cache.
2 DCLK high	Data written to merge buffer.
2 DCLK low	
3 DCLK high	Data transferred to write buffer.
3 DCLK low	
4 DCLK high	DCLK source switches to MCLK (if not already MCLK). Request sent to BIU.
4 DCLK low	
5 DCLK high	BIU receives request; if APE is high, the BIU writes the signals with address timing (A, MAS, nRW, CLF).
5 DCLK low	
6 DCLK high	BIU sets SEQ and clears nMREQ; if APE is low, the BIU writes the signals with address timing.
6 DCLK low	
7 DCLK high	BUI clears SEQ and sets nMREQ. Sets D pins.
7 DCLK low	
8 DCLK high	Buffer stage told result of write (success or abort). BIU ready to start next read or write.
8 DCLK low	
9 DCLK high	D pins continue to be driven for hold time.
9 DCLK low	
10 DCLK high	Buffer stage of pipe ready for next instruction.
10 DCLK low	
11 DCLK high	DCLK source switches to CCLK if no other reads are in progress, and clock switching is enabled.
11 DCLK low	

3.4.10 Swaps Missing the Cache

The read stage of a swap will flush the merge buffer into the write buffer and cause all outstanding writes in the write buffer to be treated as high-priority write requests.

3.4.11 Data Side Coprocessor Access Timing

3.4.11.1 System Coprocessor Register Reads

These happen when the MRC instruction is in the buffer stage, and require only a single cycle in the buffer stage.

3.4.11.2 System Coprocessor Register Writes

A request to clean a Dcache entry stalls in the buffer stage until there are 3 free entries in the write buffer. It then stalls for a further 4 cycles during which the Dcache entry is written to the write buffer. If the address is not in the Dcache or the Dcache line is clean, a cast-out bubble is introduced, resulting in a single stall cycle.

A request to drain the write buffer pushes the merge buffer into the write buffer, and then stalls until the write buffer is empty.

All other writes to the system coprocessor happen either when the MCR instruction is in the buffer stage or early in the following cycle. These writes do not stall the buffer stage.

3.4.12 Write Buffer Timing

A request to write a new write buffer entry will first be passed to the BIU on an MCLK rising edge following the second MCLK falling edge after the entry is placed in the write buffer. This applies equally to buffered and “unbuffered” writes.

3.5 The Bus Interface Unit

3.5.1 Bus Accesses and Cycles

A bus access consists of one idle cycle followed by 1,2,3,4, or 8 sequential cycles. The types of requests for bus accesses made to the BIU are:

- A single word (or part word) read request from either the data or the instruction memory system. This will result in an access containing one sequential cycle.
- A cache line fill request from either the data or instruction memory system. This will result in a bus access containing 8 sequential cycles.
- A write of a dirty cache line. This will result in a bus access containing 4 or 8 sequential cycles. The BIU treats a cache line write as a single access even if it fills two consecutive write buffer entries.
- Any other write. All other writes will write a single write buffer entry, and will require 1 to 4 sequential cycles.

Note that on reads, the final word of data is not read until the falling edge of the idle cycle following the final sequential cycle.

3.5.2 Bus Contention Resolution

The BIU will look for requests for new external accesses from the instruction and data memory systems on any cycle on which it is idle. Four types of requests are possible. These are:

- Data system read requests
- Instruction system read requests
- Normal write buffer write requests
- High-priority write buffer write requests

If there are no high-priority write requests outstanding, then the BIU will accept, in order:

- Data system read requests
- Instruction system read requests
- Normal write buffer write requests

If, however, there is both a high-priority write request and a data system read request waiting, then the high-priority write request will be performed first. High-priority write requests will only be performed before instruction system read requests if there is also a data system read request waiting.

Support, Products, and Documentation

If you need technical support, a *Digital Semiconductor Product Catalog*, or help deciding which documentation best meets your needs, visit the Digital Semiconductor World Wide Web Internet site:

<http://www.digital.com/semiconductor>

You can also call the Digital Semiconductor Information Line or the Digital Semiconductor Customer Technology Center. Please use the following information lines for support.

For documentation and general information:	
Digital Semiconductor Information Line	
United States and Canada:	1-800-332-2717
Europe:	+44 118 920 3972
Other Countries:	1-510-490-4753
Electronic mail address:	semiconductor@digital.com

For technical support:	
Digital Semiconductor Customer Technology Center	
Phone (U.S. and international):	1-508-568-7474
Fax:	1-508-568-6698
Electronic mail address:	ctc@hlo.mts.dec.com

Digital Semiconductor Products

To order StrongARM-related and other Digital Semiconductor products, contact your local distributor. The following tables list some of the products available from Digital Semiconductor:

Chips	Order Number
Digital Semiconductor 21285 SA-110 core logic chip	21285-AA

Evaluation board kits include an evaluation board, and can include a complete design kit, an installation kit, or an accessories kit.

Evaluation Board Kits	Order Number
EBSA-285 (SA-110/21285) evaluation board kit	21A85-01
EBSA-BPL-5V PCI development backplane – 5V signalling	21A85-02
EBSA-BPL-3V3 PCI development backplane – 3.3V signalling	21A85-03

Design kits include full documentation and schematics. They do not include evaluation boards or related hardware.

Design Kits	Order Number
SA-110/21285 Hardware Developer's Kit	QR-21A85 11

Digital Semiconductor Documentation

The following table lists some of the available Digital Semiconductor documentation.

Title	Order Number
Digital Semiconductor 21285 Core Logic for SA-110 Microprocessor Data Sheet	EC-R4CHA-TE
Digital Semiconductor SA-110 Microprocessor Technical Reference Manual	EC-QPWLC-TE
The StrongARM/ARM Technical Document Kit (includes the SA-110 Technical Reference Manual and the ARM Architecture Reference Manual ¹)	QR-ARMKT-TE

¹ Also available from independent booksellers: publisher, Prentice-Hall; ISBN 0 13 736299 4.

Third-Party Documentation

You can order the following third-party documentation directly from the vendor:

Title	Vendor
PCI Local Bus Specification, Revision 2.1 PCI Multimedia Design Guide, Revision 1.0 PCI System Design Guide PCI-to-PCI Bridge Architecture Specification, Revision 1.0 PCI BIOS Specification, Revision 2.1	PCI Special Interest Group U.S. 1-800-433-5177 International 1-503-797-4207 FAX 1-503-234-6762
IEEE title	The Institute of Electrical and Electronics Engineers, Inc. U.S. 1-800-701-4333 International 1-908-981-0060 FAX 1-908-981-9667