A computer system for model helicopter flight control

technical memo Nr. 1: the hardware core

Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Computersysteme

Niklaus Wirth

A Computer System for Model Helicopter Flight Control

Technical Memo Nr. 1: The Hardware Core

January 1998

A Computer System for Model Helicopter Flight Control

Technical Memo Nr. 1

The Hardware Core

N. Wirth 15.12.97

Abstract

This memorandum is the first in a series giving an account of the design and structure of the on–board computer system for controlling a model helicopter. The aircraft itself is designed by a research group of the Institut für Mess– und Regeltechnik of ETH Zürich. The computer is to stabilize the helicopter and to execute commands for flight movements. It receives inputs from various sensors and drives the necessary servos. The computer is built around an ARM processor and an FPGA.

Contents

1. Introduction

The system described in this memo was designed specifically for stabilizing the motions of a model helicopter, and for controlling its pre–programmed flight plan. On the one side stood the desire for a fairly high computing power afforded by a modern microprocessor, on the other the demand for low power consumption from batteries. The latter significantly contribute to the load (weight) of the vehicle, which should be kept as low as feasible without shortening the flight duration unduly. The outputs of the system control the various servos. Two servos control the nick and roll movements of the helicopter, one the gear (rotation about the vertical axis), one the pitch of the main rotor, and one the throttle of the engine. Inputs are received by sensing three gyros and three acceleration sensors, and, at lower and irregular rates, from a compass and from the global positioning system GPS. The system must exhibit characteristics of an embedded real–time system, and the memory must be large enough to buffer logging data. The overall composition is shown in Fig. 1.

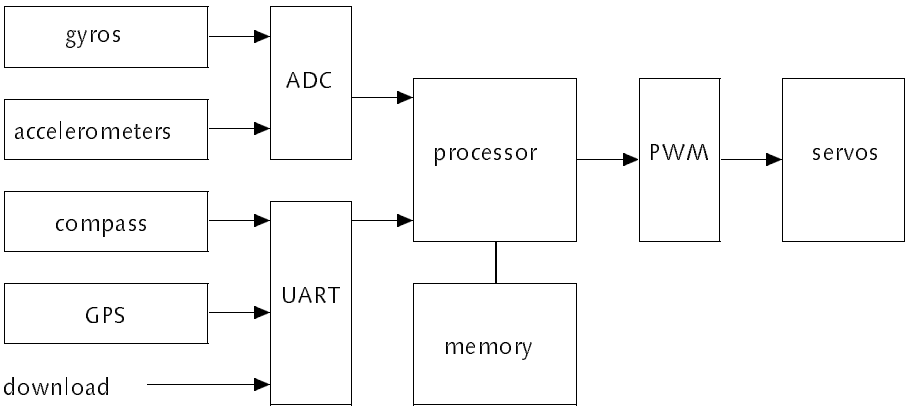

Fig. 1. Data flow

As processor the StrongARM (DEC) was chosen, primarily because of its optimal performance vs. power consumption ratio. It delivers more than 100 Mips for only 1 Watt. Also, it features an appealing, regular architecture with a modern RISC structure. For the implementation of the interface to the various peripheral devices, a programmable gate array was chosen. It allows the greatest degree of flexibility for further extensions and for modification without change of physical parts. Although serial data communication can easily be realized with this resource, an additional UART was included for this prototype. The schematics of the entire system consist of 5 pages.

Page 1 shows the processor (DC1035), the address latch (3 573s), the address multiplexer (2 157s), the memory controller implemented in a programmable device (PLD, AMD MACH211SP), and 2 "bus switches" (3384) for connecting 5V with 3.3V devices.

Page 2 contains the 4–channel UART (SN28L194), two level shifters (MAX3232) for adapting to the (outdated) RS–232 standard, and two 6–channel AD converters (196).

Page 3 shows the memory, consisting of 8 MByte of RAM and 1 MByte of ROM. The ROM can be electrically programmed from the processor (Flash–ROM).

Page 4 collects miscellaneous devices, including the power converters (MAX787 and 788) and the input filters for the ADCs. The power devices are switchers, yielding a high efficiency and minimal power loss when transforming the battery voltage down to the required levels of 5, 3.3, 3.0, and 1.8 V.

Page 5 contains the FPGA (XC6200), primarily used for generating the pulse–width modulated signals for the servos (PWM).

2. Clocking and Reset

The system derives all clock signals from a single crystal oscillator running at 3.6864 MHz. The processor internally generates a 200MHz clock and, dividing by 4, outputs the clock for the memory interface. The RAM consists of synchronous DRAM parts allowing for a high burst rate in conjunction with the processors cache buffers. Again dividing by 4, the clock is generated operating the UART and the FPGA (see Fig. 2).

The reset signal is generated by a reset button and a reset controller chip (7705). It acts as an asynchronous reset for processor, memory controller, UART and FPGA.

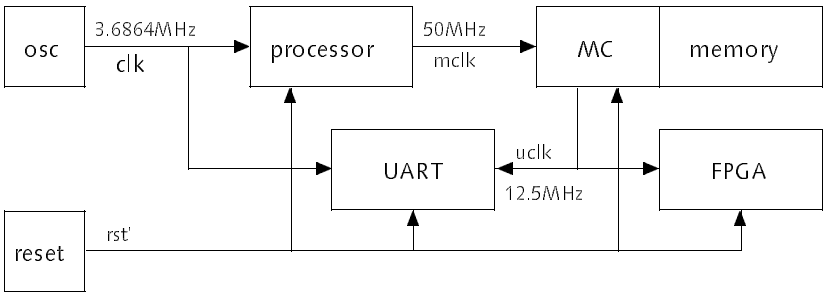

Fig. 2. Clock and reset signal distribution

3. Interrupts

The ARM processor features two separate inputs for interrupts. The "regular" interrupt (IRQ) is generated by the FPGA in 5ms intervals. It triggers the sensing of the inputs. After collecting four sets of inputs, output signals are computed to control the servos every 20ms (50Hz).

The second, fast interrupt (FIQ) is generated by the UART and affords lowest overhead for buffering received data. This is due to the fact that the processor uses a second set of registers, thereby avoiding the necessity of saving and restoring registers upon interrupt.

4. The Memory Interface

The interface between the processor and memory essentially consists of a finite state machine implemented with a programmable logic device (PLD). Addresses are time–multiplexed. a24 and a25 determine the target of a reference. If the RAM is selected, a22 serves to select one of the 2 RAM banks. Each bank consists of 2 1M x 16 SDRAM chips. a21 is a chip internal bank selector. a10 – a20 serve as the row address, and a2 – a9 as the column address. a0 and a1 are the byte selectors.

Address assignment

a25/24/22	target	address range	
0––	RAM	0000 0000 – 003F FFFF	(4M)
100	ROM	0200 0000 – 020F FFFF	(1M)
101	RAM command	024x xxxx	
110	I/O devices	0300 0000 – 033F FFFF	

The memory controller and the SDRAMs operate with the memory clock delivered by the processor at 50 MHz (20ns cycle time). The interface is shown in Fig. 3.

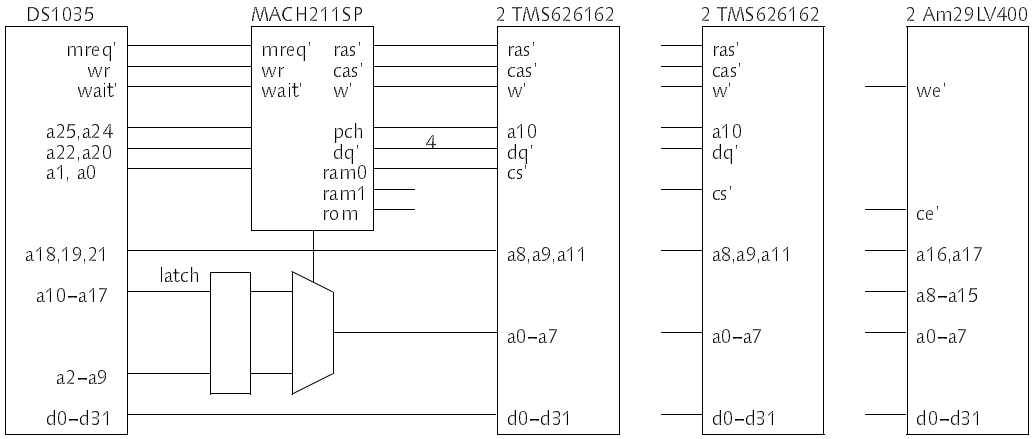

Fig. 3. The memory interface

The controller essentially is a state machine (see Fig. 4). In its neutral state 0 it waits for a request. Each state is indicated by a box containing the state's identification. The names in the boxes denote the signals that are active in the designated state. Note that a name with an apostrophe denotes an *active low* signal. Signal names along edges indicate (input) values that determine the next state. Names on top of boxes denote a RAM command.

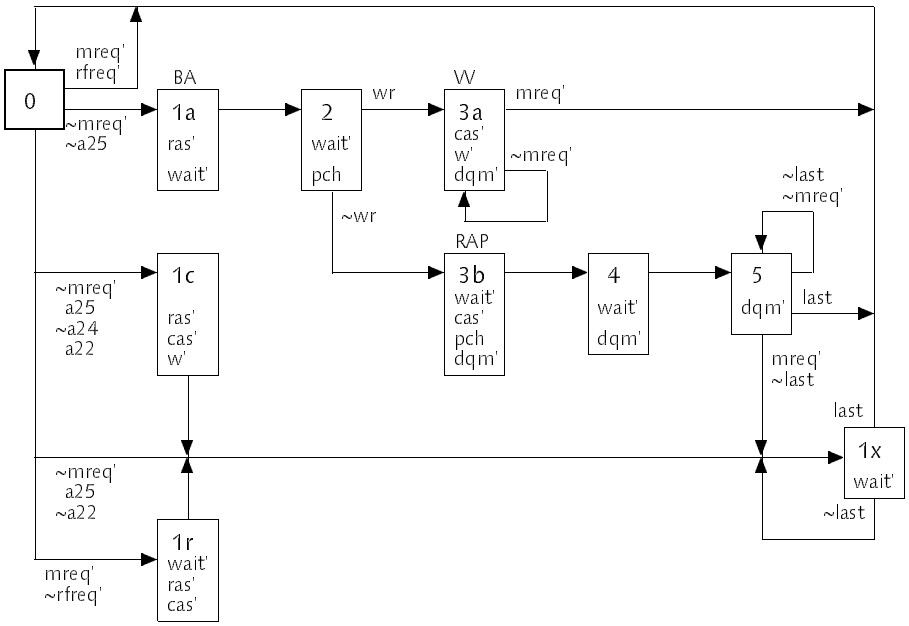

Fig.4. The state machine

The controller being in neutral state 0, the processor initiates a memory access by asserting *mreq'*. At this time *wr* indicates whether it is a read or a write operation, and *a24/5* specify the target. For a write–ram cycle, the machine goes through states 1a, 2, and 3a, for a read–ram cycle, states 1a, 2, 3b, 4, 5, and for other cycles through states 1x, 1c, or 1r. States 5 and 1x may be repeated a given number of times. For this purpose, in addition to the state machine, there exists a *counter.* This counter delivers the signal *last* which terminates repetition and deactivates the counter. The counter is triggered in the following transitions:

0 –> 1x	ROM access	8 cycles
0 –> 1x	I/O access	16 cycles
0 –> 1c	SDRAM command	8 cycles
4 –> 5	SDRAM read	8 cycles

Write cycles: The SDRAM operates with a cas latency of 2 and a burst length of 1. This implies that cycle 1a, issuing a bank activate command to the SDRAM, must be followed by 2 wait cycles, i.e. cycles holding the processor by activating *wait'*, before a write command is issued in state 3a The machine repeats state 3a until *mreq'* becomes inactive; then a lst write command is issued which also deactivates the accessed bank. Note that the processor delivers data one cycle after the address; data are sampled by the RAM in the cycles where *dqm'* is active. The signal flows are shown in Fig. 5.

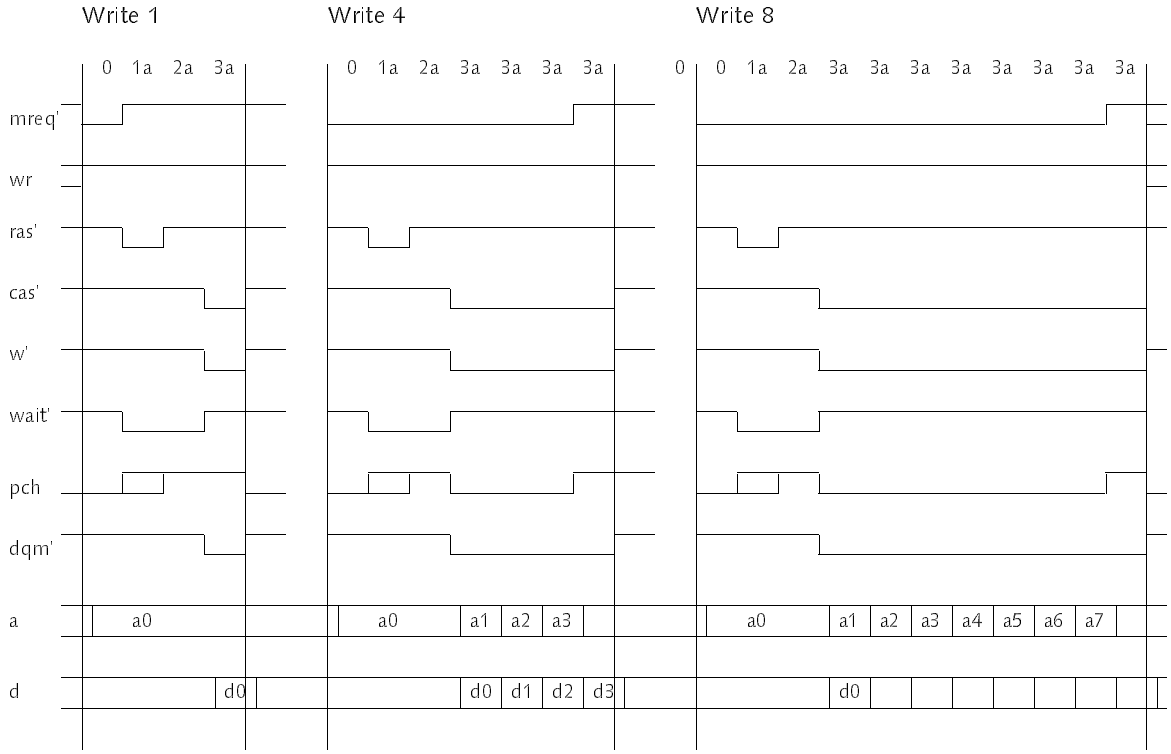

Fig. 5. Write cycles of lengths 1, 4, and 8

Read cycles: The SDRAM operates with a cas latency of 2 and a burst length of 8. A burst length of 1 would be inappropriate, because the requests from the processor terminate two cycles ahead. During a burst the word addresses are generated by the RAM internally (a0–a2). After the 2 wait states (1a, 2), 2 more follow (3b, 4). A read (with bank dectivation) is issued in state 3b, and the data are sampled during 8 states 5, the counter having been triggered when entering 5 the first time. If a read request is for less than 8 words, the read burst continues for the full 8 word stream. But the processor is held by asserting *wait'*, and the RAM output is suppressed by deactivating *dqm'*. The signals flows are shown in Fig. 6.

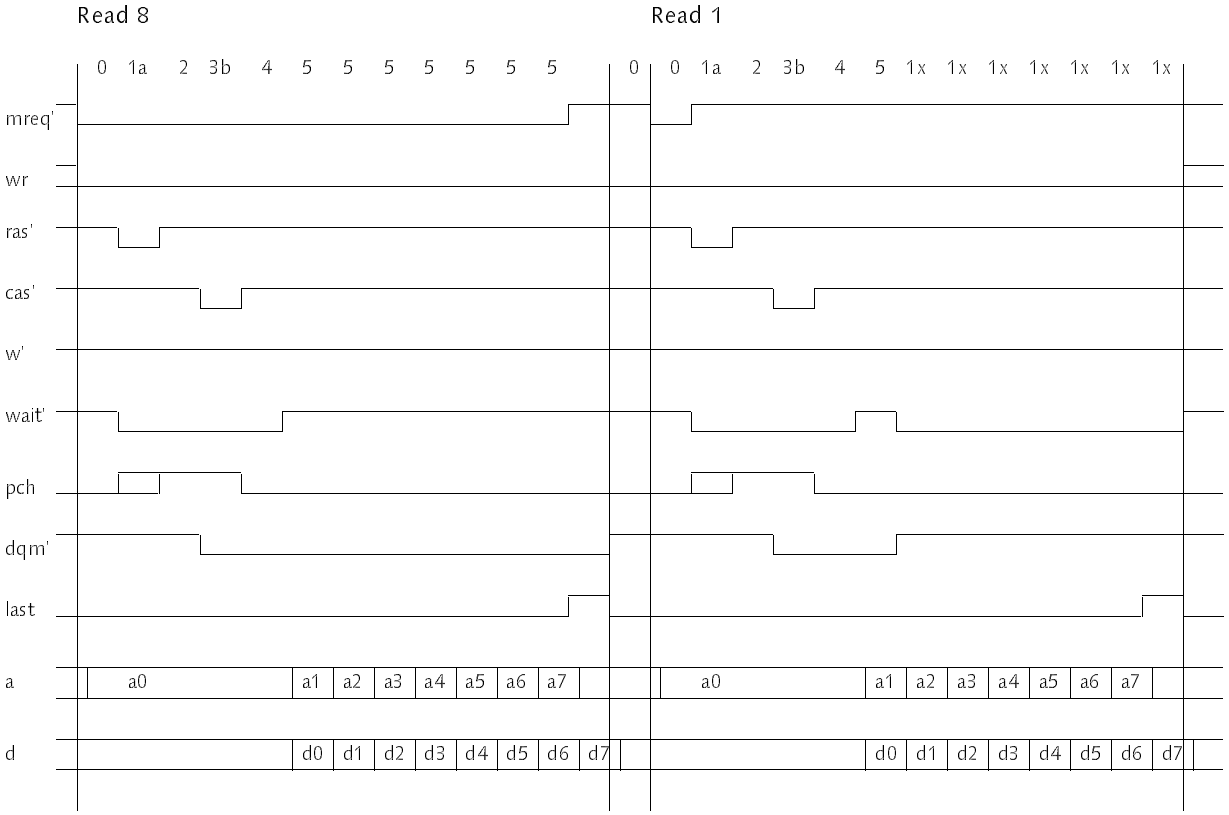

Fig. 6. Read cycles of lengths 8 and 1

Non–RAM cycles: These are specified by a25 = 1. The state machine activates *wait'* to hold the processor and triggers the counter (state 1x). If both a25 and a24 are 1, the counter runs through 16, otherwise through 8 cycles before it activates *last*, yielding accress cycles of 160ns for the ROM and of 320ns for I/O devices.

RAM commands: The SDRAM must be initialized properly upon system startup time. This is done by a write request with a25 = 1, a24 = 0, a22 = 1. The state machine acts like for a non–ram request, with the exception that for one cycle it activates *ras'*, *cas'* if a0 = 1, and *w'* if a1 = 1 (state 1c). The signals determine the ram command as follows:

ras'	cas'	w'	command
0	0	0	set ram mode
0	0	1	auto–refresh
0	1	0	stop burst (not used)

In the set mode command, the values of the addresses specify the various parameters as follows:

proc adr	ram adr	property	adr field value	property value
a10–a12	a0–a2	read burst length	0	1
			1	2
			2	4
			3	8
a13	a3	burst type	0	serial
			1	interleaved
a14–a16	a4–a6	cas latency	1	1
			2	2
			3	3
a19	a9	write burst length	0	= read burst length
			1	1

RAM refresh: The same action as for the Refresh command is also triggered by the *rfreq'* signal becoming active. It becomes effective only in state 0 and if *mreq'* is inactive. Hence, refresh requests cannot interrupt an ongoing access operation. State 1r forces *cas'* = 0 and *w'* = 1, and it is followed by 7 wait cycles. *rfreq'* is set low when the 10–bit *refresh counter c* reaches the value 0. Hence, a refresh cycle is inserted every 1024 * 20ns = 20.5us.

The design of the state machine is such that the output signals are directly generated by registers that also determine the state. This solution yields the sharpest edges for the outputs. Note that *ras'*, *cas'*, *w'* determine the RAM command, and that all three being 1 means no RAM action. In these cases, *wait'*, *pch, and dqm'* are also used to distinguish between states. Furthermore, *ras'* controls the address multiplexer; when low, the upper bits (row adr) are fed to the RAM and *pch = a20.* When a read or write command is issued to the RAM, *pch* determines whether a bank deactivation through precharge should follow the read or write operation. The state assignment is shown in Fig. 7, where those signal values are printed in boldface that are relevant for the state's unique definition.

state	ras'	cas'	w'	wait'	pch	dqm'	
0	**1**	**1**	1	**1**	0	**1**	idle
1a	**0**	**1**	**1**	0	a20	1	bank select
2	**1**	**1**	1	**0**	**1**	1	
3a	**1**	**0**	**0**	1	mreq'	0	write
3b	**1**	**0**	**1**	0	1	0	read
4	**1**	**1**	1	**0**	**0**	**0**	
5	**1**	**1**	1	**1**	0	**0**	
1m	**0**	**0**	**0**	0	1	1	mode command
1r	**0**	**0**	**1**	0	1	1	refresh command
1d	**0**	**1**	**0**	0	1	1	deac command
1x	**1**	**1**	1	**0**	**0**	**1**	wait

Fig. 7. State assignment

The counter is chosen not as a binary down–counter, but as a sequencer with a state assignment shown in Fig. 8. The reason for this is the need for fewer and–terms for counter values s1, s2, and s3. The idle state of the counter is with all register values = 0. The counter is triggered by setting s0 to 1 (and, in the case of an I/O request, also s3 to 1).

state	s3	s2	s1	s0	last
0	1	0	0	1	
1	1	0	1	1	
2	1	0	1	0	
3	1	1	1	0	
4	1	1	1	1	
5	1	1	0	1	
6	1	1	0	0	
7	1	0	0	0	
8	0	0	0	1	
9	0	0	1	1	
10	0	0	1	0	
11	0	1	1	0	
12	0	1	1	1	
13	0	1	0	1	
14	0	1	0	0	
15	0	0	0	0	1

Fig. 8. Counter variable assignment

From the state definitions and the transition diagram (Fig. 4) the inputs of the state registers can be derived in the form of sums of products, as is most suitable for PLD–implementations. This is shown in Fig. 9, where each row denotes a term with the corresponding transition indicated at the right end of the page. In the same manner, Fig. 10 illustrates the transitions of the counter registers.

variable	ras'	cas'	w'	wait'	pch	dqm'	last	wr	a25..	a0	m/rfreq'	transition
ras'	1	1		1		1			0–––		0	0 – 1a
	1	1		1		1			101–		0	0 – 1mrd
	1	1		1		1					10	0 – 1r
cas'	1	1		0	1							2 – 3ab
	1	0	0		0						0	3a – 3a
	1	1		1		1			101–	–1	0	0 – 1mr
	1	1		1		1					10	0 – 1r
w'	1	1		0	1			1				2 – 3a
	1	0	0		0						0	3a – 3a
	1	1		1		1			101–	1–	0	0 – 1md
	1	1		1		1		1	11–		0	0 – 1x
	1	1		0	0	1		1	11–			1x – 1x
wait'	1	1		1		1					0	0 – 1
	1	1		1		1					10	0 – 1r
	0	1		1					0–––			1a – 2
	1	1		0	1			0				2 – 3b
	1	0	1									3b – 4b
	1	1		1		0	0				1	5 – 1x
	0	0										1mr – 1x
	0	1	0									1d – 1x
	1	1		0	0	1	0					1w–1x
pch	1	1		1		1			0––1		0	0 – 1a
	1	1		1		1			101–		0	0 – 1mrd
	0	1	1						0–––			1a – 2
	1	1		0	1			0				2 – 3b
dqm'	1	1		0	1							2 – 3ab
	1	0	0		0						0	3a – 3a
	1	0	1									3b – 4
	1	1		0	0	0						4 – 5
	1	1		1		0	0				0	5 – 5

Fig. 9. State transitions

variable	ras'	cas'	wait'	pch	dqm'	s3	s2	s1	s0	a25	m/rfreq'	transition
s3	1	1	1		1					11–	0	0 – 1rom
						1	1					
						1		1				
						1			1			
s2						0	1	0				
						1	1					
						1	0	1				
s1						0	0	1				
						0	1					
						1	1	0				
s0	1	1	1		1					1––	0	0 – 1xmrd
	1	1	1		1						10	0 – 1r
	1	1	0	0	0							4 – 5
							0	0	1			
							1	1				
						1	0	0	0			
last						0	1	0	0			

Fig. 10. Counter transitions

A last consideration concerns the startup. Upon reset, the processor starts fetching an instruction at address 0. As the boot sequence is loaded into ROM, and as the lower part of the address space was allocated to RAM, an address translation must be inserted upon startup. This involves the signal *boot*, reset to 1, and set to 0 as soon as the first memory request has been terminated. If *boot* is active, ram requests are suppressed and instead interpreted as rom requests. Effectively, this results in the first

RAM–request being diverted to ROM.

The Lola specification for the memory controller is directly derived from the transition tables of Fig. 9 and Fig. 10.

5. RAM Initialization

The first instruction, executed with boot = 1, must be a branch to the ROM, where the boot program resides, i.e. to address 2000004H. Then the RAM must be initialized with the following sequence of write instructions with special addresses (a25 = 1, a24 = 0, a22 = 1):

2400002H	deactivate banks (precharge)
2400001H	auto refresh, issue 8 of these instructions
2488C03H	set mode register: write burst length 1, latency 2, serial, read burst length 8, or
248AC03H	set mode register: write burst length 11, latency 2, interleaved, read burst length 8