ATC0 Engineering Document

Krste Asanović
Kenneth Barr
Albert Ma
Michael Zhang

Contents

List of Figures

1 Introduction

ATC0 (ASSAM Test Chip #0) is the first test chip designed by the SCALE group at MIT. It serves the purpose of debugging the design process, tool chain flow, testing strategies for future hardware modules, and characterizing the fabrication process. ATC0 also implements an interface between itself and an external controller which will be used for the series of future Pekoe chips. ATC0 will be fabricated in a $0.18\,\mu$m CMOS process with six aluminum metal layers from TSMC, sized at 2.7 mm$\times$2.7 mm. Figure 1 shows the overview of ATC0.

ATC0, which we will refer to as the *chip*, is controlled by a *host* platform. The communication between the host and chip is accomplished with an asynchronous handshake protocol (*AHIP*) described in section 2. While any host implementing this protocol can interface with ATC0, the intended host is currently comprised of a personal computer (PC) and the *Assam Tester Baseboard*. ATC0 will sit on a daughter card which plugs into the baseboard.

ATC0 contains four major components: asynchronous host interface port (*AHIP*); test harness (*tester*); clock generator (*clockgen*); device-under-tests (*DUTS*). There are three DUTs on ATC0. DUT0 is a 1KB low-power SRAM block developed within our group. DUT1 and DUT2 are two commercial SRAM blocks sized at 1KB and 4KB. AHIP implements the chip side of the asynchronous protocol. Tester contains a pattern generator and a logic analyzer, both 4KB SRAM blocks. The pattern generator holds microcode to be executed to test the DUTs. The logic analyzer holds outputs from the DUTs. Clockgen is controlled by an external analog voltage to produce a continuous range of clock frequency up to 1 GHz. The tester is clocked at the half of the frequency of the DUT clock for reliability. There is a 112-bit bus carrying test vectors from test harness to DUT a 128-bit bus carrying DUT output from DUT to test harness.

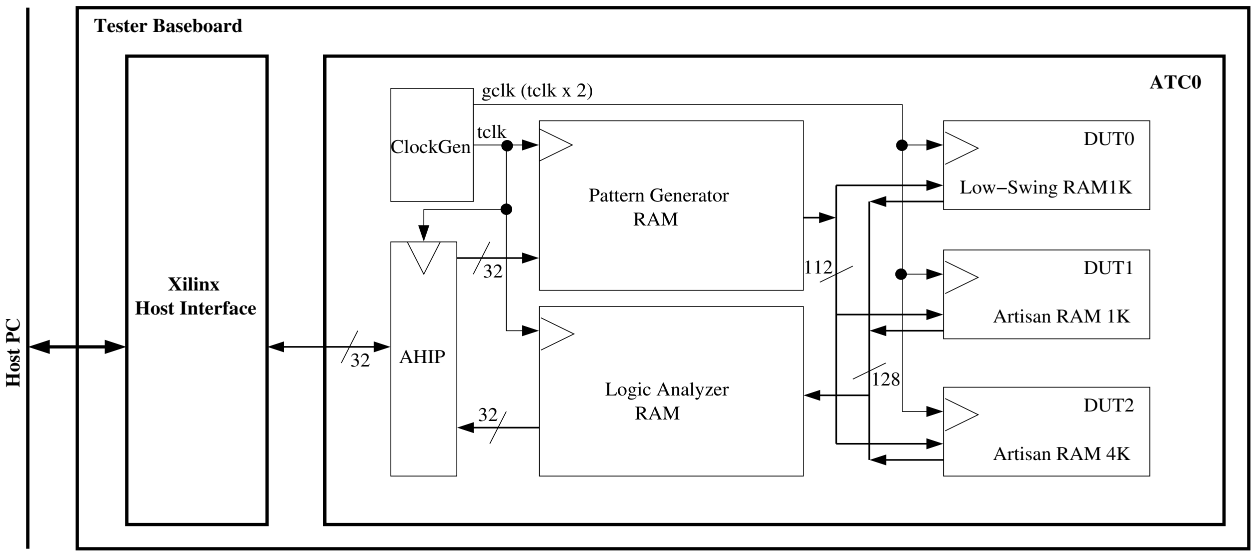


Figure 1: Test chip overview.

2 Asynchronous Host Interface Port

AHIP is the data communication channel between the host and ATC0, through which the data travels through a 32-bit wide bi-directional bus. ATC0 is a slave device to the host. The host and ATC0 run on asynchronous clocks, using two bits, `req` and `ack`, to implement a handshake protocol. The host runs at the speed of the PCI bus (33 MHz) and we will refer to this clock as the host clock, `hclk`. ATC0 can be run at a wide range of frequencies, asynchronous to the host, which we will refer to as the `tclk`.

2.1 Asynchronous Transaction Protocol

31 28	27 24	23 0
opcode	bmc	address
4	4	24

Figure 2: Header word format.

Opcode	Function
0000	Normal Word Write
0001	Normal Word Read
1000	Test Word Write
1001	Test Word Read (Data)
1101	Test Word Read (Address)

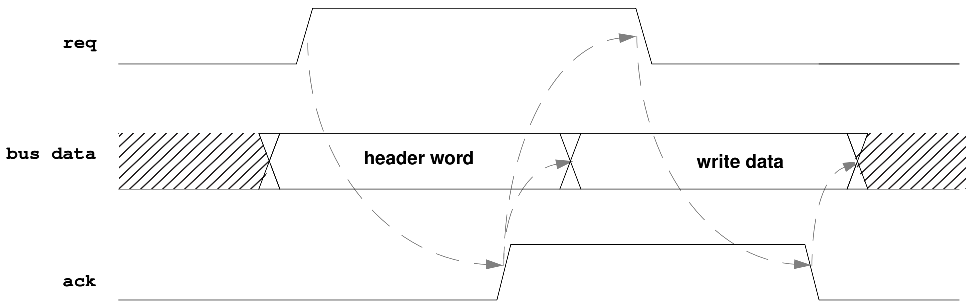
Table 1: Opcodes for the AHIP.

The AHIP protocol is done through two handshake signals, `req`, controlled by the host, and `ack`, controlled by the chip. When in *idle* state, both `req` and `ack` are low. The host starts a transaction by raising the `req` signal and placing a *header word* on the bus. The format of the header word is shown in Figure 2. The header word contains three fields: a 4-bit opcode field; a second 4-bit which contains a *burst-mode counter* (`bmc`) for burst-mode read and write; and a 24-bit address. The burst-mode read/write will not be implemented on ATC0 while `bmc` field is reserved for future implementations. The value in the `bmc` field is ignored. Table 1 shows the different opcodes. During normal operation, only the normal word read and normal word write are used. Some opcode are used to perform self-test of AHIP, as described in Section 2.4.

2.2 Writing and Reading to ATC0

The host starts a transaction by placing the header word on the bus and raising the `req` signal. After the chip observes that `req` is high, it reads the header word and raises the `ack` signal. After the host receives `ack`, it places the write data on the bus and lowers `req`. The chip reads the bus value and then lowers `ack`. The host will then free the data bus and both the host and the chip return to idle state. Figure 3 shows the timing diagram of the transaction. While AHIP is in IDLE state, the host is responsible for drive values onto the bus, preventing it from floating.

Similar to word write, the host starts the transaction by placing the header word on the bus and raising the `req` signal. The chip will obtain the header and asserts `ack`. Simultaneously, the read operation is performed and output data is temporarily placed in an output buffer. Once the host sees the `ack`, it frees the data bus and lowers `req`. The chip then places the data on the bus and raises `ack`. The host reads the data off the bus and raises `req` again, the chip then frees the data bus and raises `ack`. The host lowers `req` and followed by the chip lowering `ack`, returning to idle state. Figure 4 shows the timing diagram of the word read. Similarly, the host is responsible for driving the bus while AHIP is IDLE.



(a)

Figure 3: Timing diagram of word write.

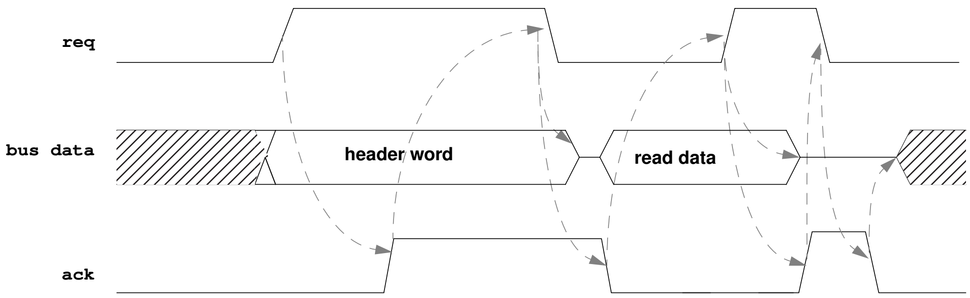


Figure 4: Timing diagram of word read.

2.3 AHIP Interface

Figure 5 shows the AHIP implementation. AHIP communicates with the host through a 32-bit data bus and three control signals. Signal `ext_rst` is used to reset the AHIP. When resetting, `ext_rst` must be held high for at least 5 `tclk` cycles. Signals `ext_req` and `ext_ack` are used for the handshake protocol. Data, `ahip[31:0]`, are passed through a bi-directional pin. Signal `ahip_en` is the signal to enable tristate drive to drive the output bus. Singal `ext_rst` is synchronized through two registers to minimize the probability of meta-stability failure.

AHIP communicate to the rest of the chip through six signals. Signal `ahip_core` is the synchronized 32-bit data from AHIP to ATC0 core; signal `core_ahip` is the output data from ATC0 core to AHIP. AHIP can issue either a write request, `w_req`, or a read request, `r_req`, with an associated address, which is the lower 24 bits of `cmd`. After a request is issued, AHIP waits for the `done` signal.

The AHIP controller, `AHIP CONTROL`, generates the necessary control signals. Signal `cmd_en` is used to store address and opcode fields into registers when the data word being processed is a head word. Signal `bus_en`

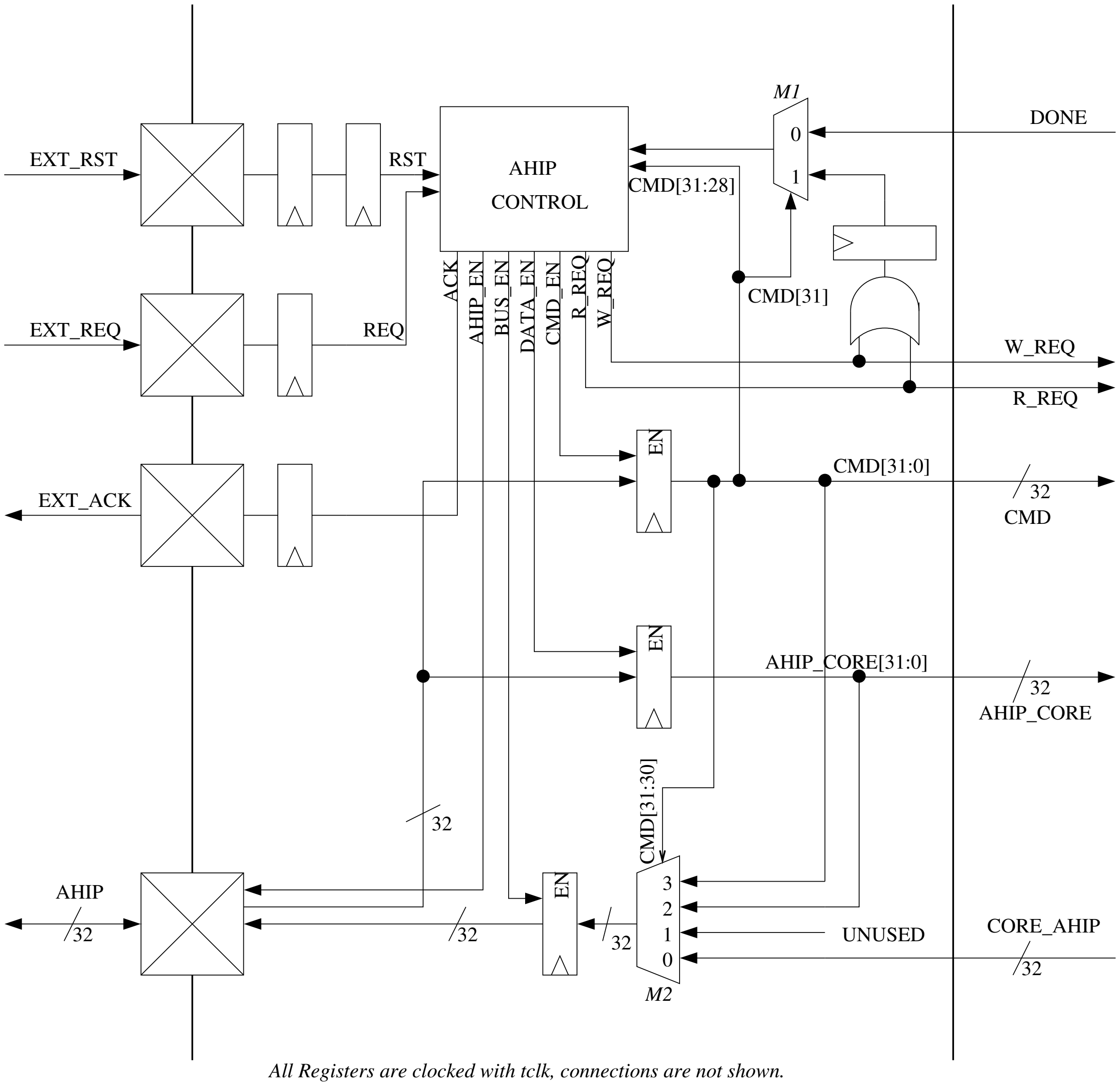


Figure 5: Block diagram of the AHIP.

is used to store output data when it is ready. Signal `ahip_en` is used to enable the bi-directional data pad when driving data from core to AHIP. Signal `ack` is the handshake acknowledgement. Finally, `r_req` and `w_req` can be issued to the appropriate devices on the chip. The control observes the `done` bit to determine whether the read or write request is finished.

2.4 AHIP Self-Test

To verify the correctness of AHIP functionality, the top bit of the opcode is used to run AHIP in test mode. Refer to Figure 5, multiplexer *M1* is used to mux the `done` signal back to the AHIP control signal. When in test mode, the done signal is generated by ORing the `r_req` and `w_req`, as one of them is asserted under correct AHIP operation. Thus, the `done` signal will be asserted two cycles later. Multiplexer *M2* is used to mux back the data. When in normal mode, `core_ahip` will be muxed back as desired. When in test mode, the host can use the second most significant bit of the opcode to choose to read back either the last written data, or the current read address, verifying both the data and the address path. An extra register is used to keep a copy of the last write data, which is enabled by `w_req`, which is asserted during a correct write operation.

3 Clocking

3.1 Clock Generation

Figure 6 (a) shows the overview of clock generation for the testchip. `Gclk` is the clock used for all the DUTs `Tclk` is the clock used for the AHIP and the test harness. It is a divided by two version of `Gclk`. `Pclk` is fed to an output pad in order to measure the clock frequency. It is 1/16 of the frequency of `Gclk`, and is generated by a cascade of frequency dividers.

The clock generator operates in three modes, controlled by the `fsel[1:0]` signal. On `fsel=0`, the clock is generated from an internal voltage-controlled oscillator (VCO). The frequency of the VCO is controlled by an offchip analog voltage `VC`. The VCO generates a differential clock which is converted to a single-ended clock. On `fsel=1`, the clock is taken directly from an offchip `Eclk`. On `fsel=2`, the clock is `Eclk` divided by 2. This serves to create a 50% duty cycle clock.

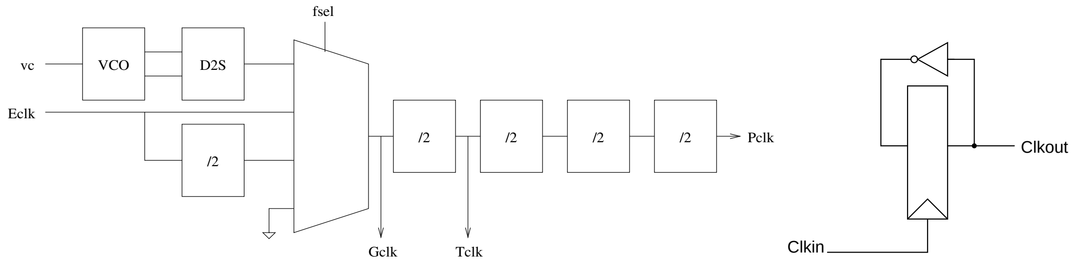


Figure 6: (a) Overview of clock generation. (b) Clock divider.

Figure 7 shows the voltage-controlled oscillator. The design is from Hwang et.al. in ISSCC 2002. It is based on a slew-limited differential inverter pair. This design is shown to have good supply noise rejection compared to basic differential current-starved inverters. In addition, the differential design generates less inductive supply noise than a single-ended design.

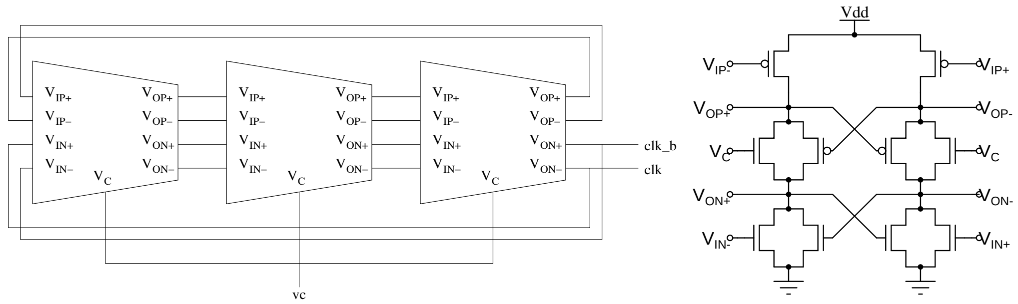


Figure 7: (a) VCO (b) differential inverter stage.

Figure 8 shows the differential to single-ended converter. It's purpose is to generate a 50% duty cycle clock without needing to divide the clock frequency by 2. It is in essense two stages of op-amps. The first stage is mirrored to balance the inputs and to provide a differential input to the second stage.

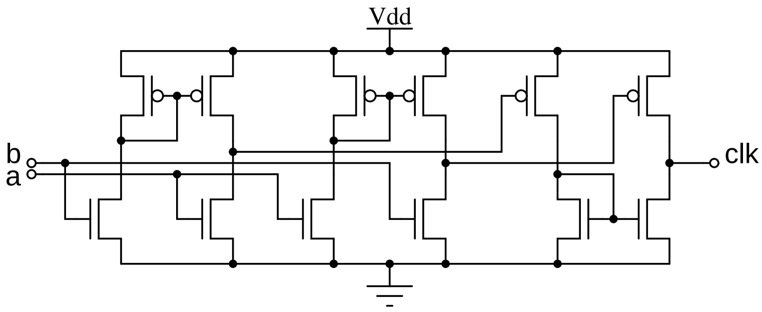


Figure 8: Differential to signal-ended converter

Figure 9 shows how the VCO's frequency depends on `VC`.

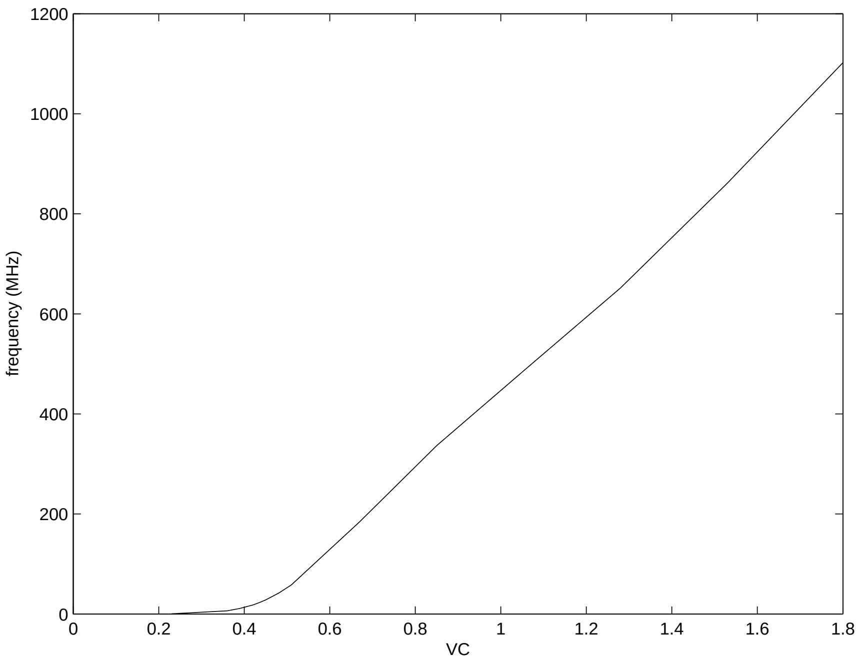


Figure 9: VCO Frequency vs voltage. Vdd=1.8V

Table 2 shows how the VCO's frequency depends on `Vdd`.

Vdd	Vc	frequency (MHz)	% change
1.62	1.53	834.0	-2.1
1.80	1.53	851.8	0
1.98	1.53	856.9	0.6

Table 2: Frequency dependence on Vdd

4 Test Harness

The test harness handles the reading and writing requests from the AHIP, and the execution path for the DUTs. The main components of test harness are: a pattern generator, which holds test code for the DUTs; a logic analyzer, which holds results from the DUTs; a control register which controls the execution of the test harness; a status register which indicates the status of the execution; and the test harness control logic. Each component of the test harness will be described in detail in the following sections. Figure 10 shows the block diagram of the test harness.

4.1 Pattern Generator and Logic Analyzer

The pattern generator (PGRAM) holds the microcode instructions for DUT testing. Logically, PGRAM has 256 entries of 128-bit each. Each entry in PGRAM contains an 8-bit *next address* field, an 8-bit *output* field, and a 112-bit *test vector* field, as shown in Figure 11. The address field contains the address of the next

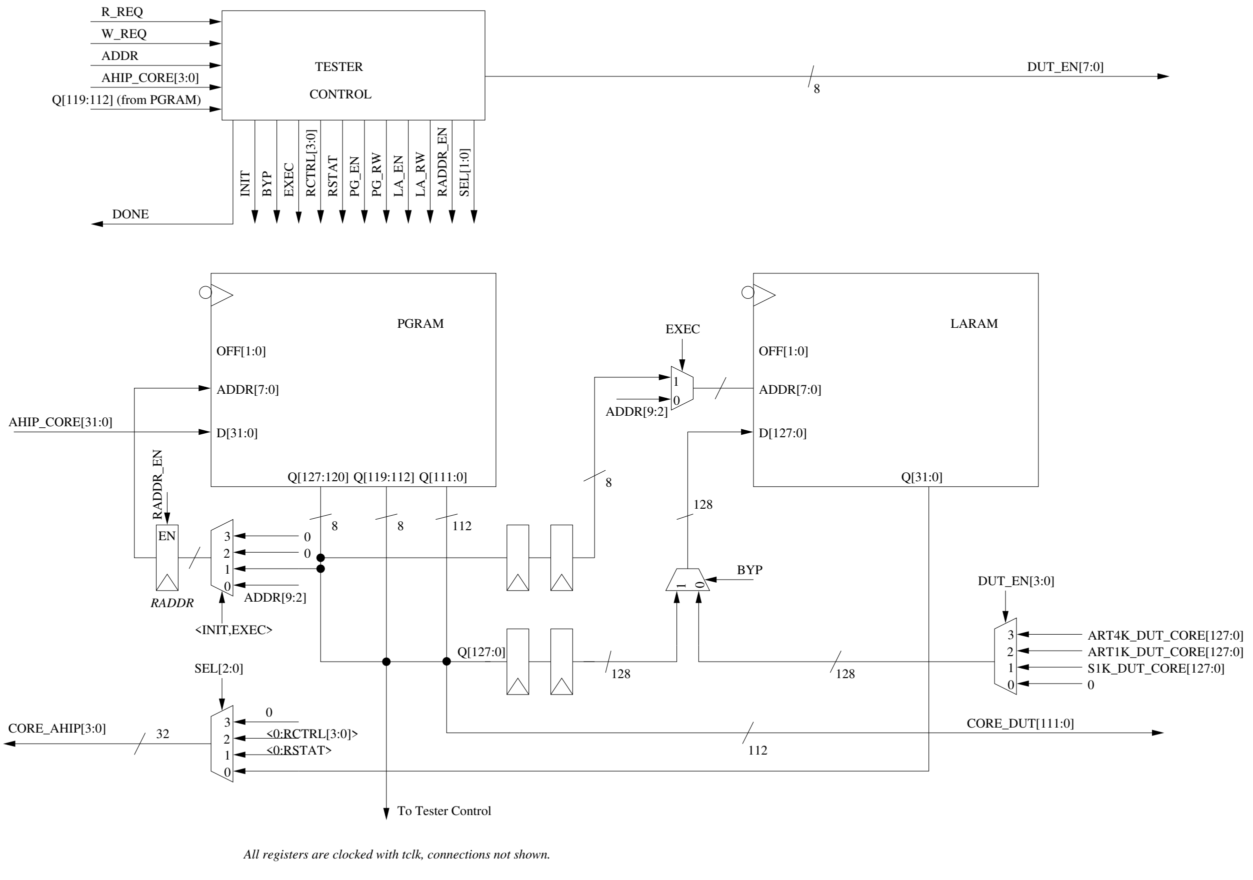


Figure 10: Block diagram of Test Harness.

instruction to be executed. The output field currently only contains a 1-bit `stop` bit which is used to halt the execution of the test harness.

127 120	119 112	111 0
next_addr	output	test vector
8	8	112

Figure 11: Pattern generator RAM format.

The Logic Analyzer (LARAM) holds the sampled output from DUTs. Logically, it contains 256 entries of 128-bit each. It captures the execution results of the DUTs. If the DUT is a simple unpipelined unit with 1 cycle latency, row *N* of the LARAM captures the results of the DUT execution of the microcode in row *N* of the PGRAM. The output format is defined specifically for each DUT.

4.1.1 PGRAM/LARAM Implementation

The test code and the results are channeled between the host and the test harness through AHIP. Since AHIP is 32-bit, writes to PGRAM and reads from LARAM are done one word at a time. However, when the test harness is in execute mode, a 128-bit entry must be read out from PGRAM and a 128-bit entry must be written to LARAM within each cycle. Therefore, both PGRAM and LARAM are implemented as four seperate SRAM banks with 32-bit width, each holding one word of the PGRAM/LARAM entries. When performing AHIP requests, only one bank needs to be activated. When executing, all four banks are activated.

4.2 Control and Status Registers

In order to control the test harness, the CPU write the necessary control signals to a control register, `$rctrl`, shown in Figure 12. The control register simply consists of two fields: the `exec` field (1-bit) is used to determine the mode of the test harness; and the `dut_en` field (3-bits), which controls which DUT to activate. For ATC0, three `dut_en` bits are used to select which DUT to enable. In future testchips, if the number of DUTS increase, we could simply extend the `dut_en` field to accomodate them. The encoding of the `dut_en` is shown in Table 3. When the test harness is in the bypass mode, it simply sends the 128-bit data read from the PGRAM to LARAM as DUT outputs. Figure 12 also shows the status register, which is simply a one-bit register indicating whether execution is done.

`$rctrl`

dut_en[2:0]	exec
3	1

`$rstat`

status
1

Figure 12: Control Register and Status Register Format.

dut_en[2:0]	DUT
000	Bypass test mode
001	1KB Low-Swing SRAM
010	1KB Artisan SRAM
011	4KB Artisan SRAM
100 → 111	Unused

Table 3: DUT enable encoding.

4.3 Test Harness Address Map

Only the lower 12-bits of the address are used by the test harness. Figure 4 shows the address map. The test harness is word addressable. Each row in the PGRAM and LARAM are big-endian.

Address	Memory
`0x000` → `0x3ff`	PGRAM
`0x400` → `0x7ff`	LARAM
`0x800`	`$rstat`
`0xc00`	`$rctrl`

Table 4: ATC0 Address Map.

4.4 Test Harness Timing

Figure 13 gives an illustration of the test harness timing. Valid test vectors are read from the PGRAM after the falling edge of the clock during each cycle. The results of the test vector is expected two cycles later, before the falling clock edge. This timing constraint gives the DUTs one `tclk` cycle time to process the test vector and generate the result. If the DUT meets this constraint, then the result of test vector in row N of PGRAM will be in row N of the LARAM. Otherwise, if the DUT takes more than 1 cycle to process the test vector. The LARAM location of the results will be shifted accordingly. For example, if DUT takes X cycles to finish, then the N'th row test vector result will be in $N-X-1$ row of the LARAM. In addition, the results of the last $X-1$ test vectors will not be collected by the LARAM.

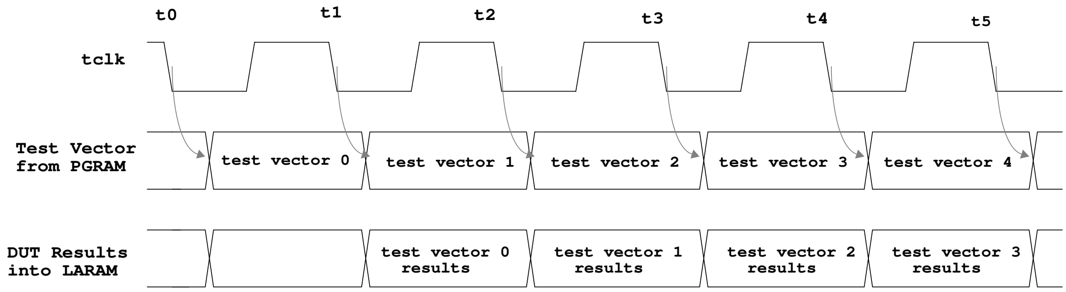


Figure 13: Test Harness and DUT Timing Diagram.

4.5 Test Harness Interface

It communicates with the AHIP through six signals: `r_req`, `w_req`, and `address` are AHIP read/write requests and its associated address; `done` is the status signal for the read and write requests; `ahip_core` and `core_ahip` are two uni-directional bus for data transfer. It communicates with the rest of chip through two data buses, and two enable signals. An 112-bit bus, `core_dut`, is the bus carrying test vectors from test harness to DUTs. After DUTs operate on the test vectors, their output is carried back to the test harness through the 128-bit `dut_core` bus. Signal `dut_en` is used to enable the DUTs. At most one DUT can be enabled at one time.

There are two basic mode of test harness: *non-exec* and *exec* mode, determined by the `exec` bit of the control register. When in *non-exec* mode, (`exec = 0`), the test harness accepts read and write request and processes them. The address of the PGRAM and LARAM address for read and write are provided by the address input from AHIP. When in *exec* mode (`exec = 1`), it will execute the microcode in the PGRAM to completion.

4.5.1 Initialization

Before using test harness, it must be initialized. This is accomplished by issuing a write to the status register with the value of zero. This will clear the status register, the control register, and the address register. This also implies that test harness will always start its execution from entry 0.

4.5.2 Termination

There are two ways to halt execution: by using the stop bit or writing to the control register. To use the stop bit to halt execution, set the stop bit of the last instruction *and* the second last instruction. Execution will halt after the last instruction and status register will have a one as output. The host can poll the status register to see whether execution finished. In situations when the stop bit cannot be used, e.g., an infinite loop, the host can issue a write to the control register which will bring the test harness out of execution mode back to idle. The test harness then will perform the write request.

5 Devices Under Test

We place 3 DUTs (device-under-test) on ATC0, though only one DUT will be enabled at any time. During each test harness cycle, an *112-bit* value that encodes two micro-instruction is read out from the PGRAM. The instruction is latched and becomes available at the following rising clock cycle of `tclk`. The DUTs are clocked at twice of the frequency of the tester. We will refer to the DUT clock as `gclk`. The clock signals are synchronous. The two micro-instructions within each 112-bit value are placed on the 56-bit bus sequentially. One bit, `datasel` (we use `tclk` as `datasel`), will be used to indicate whether the word on the bus is the upper or lower word of the micro-instruction. Each DUT has its own glue logic which decodes the bus value, and peroforms the appropriate operation.

5.1 Micro-Instruction Format

Figure 14 shows the micro-instruction format for the two 1KB SRAM blocks — S1K and ART1K. S1K is the low-swing SRAM block designed by our group. ART1K is the 1KB SRAM block from Artisan.

111 84	83 83	82 82	81 81	80 80	79 72	71 64	63 32	31 0
`unused`	`cen_hi`	`cen_lo`	`wen_hi`	`wen_lo`	`addr_hi`	`addr_lo`	`data_hi`	`data_lo`
28	1	1	1	1	8	8	32	32

Figure 14: 1KB SRAM Micro-Instruction Format.

Figure 15 shows the micro-instruction format for the 4KB Artisan SRAM block — ART4K.

111 88	87 87	86 86	85 85	84 84	83 74	73 64	63 32	31 0
`unused`	`cen_hi`	`cen_lo`	`wen_hi`	`wen_lo`	`addr_hi`	`addr_lo`	`data_hi`	`data_lo`
24	1	1	1	1	10	10	32	32

Figure 15: 4KB SRAM Micro-Instruction Format.

5.2 Output Format

For all three DUTs, the outputs resume the format in Figure 16. When writing, the outputs are random, when reading, the outputs are data stored at the specified addresses.

127 64	63 32	31 0
`unused`	`high-instr output`	`low-instr output`
63	32	32

Figure 16: DUT Output Format.

6 Pin Out

ATC0 assumes the dimension of 2.7 mm×2.7 mm. There are 84 pins in total with 40 signal pins. Pad pitch is 0.09 mm. Table 5 lists the signal names of all active pads for ATC0.

Name	Direction	Width	Description
System Reset			
EXT_RST	I	1	System reset.
System Clock			
VC	I	1	Voltage control for clock frequency.
EXT_PCLK	O	1	Clock out.
EXT_ECLK	I	1	External clock.
FSEL[1:0]	I	2	Clock frequency select.
Asynchronous Host Interface Port			
AHIP[31:0]	B	32	Data input.
EXT_REQ	I	1	Asynchronous transaction request.
EXT_ACK	O	1	Asynchronous transaction acknowledgement.
	Total	40	

Table 5: ATC0 Signal Pads

7 Packaging

ATC0 is packaged with PGA121M ceremic package. This package has 120 pins and a cavity of 325 Mil. Figure 17 shows the physical connection of the pins to the package.

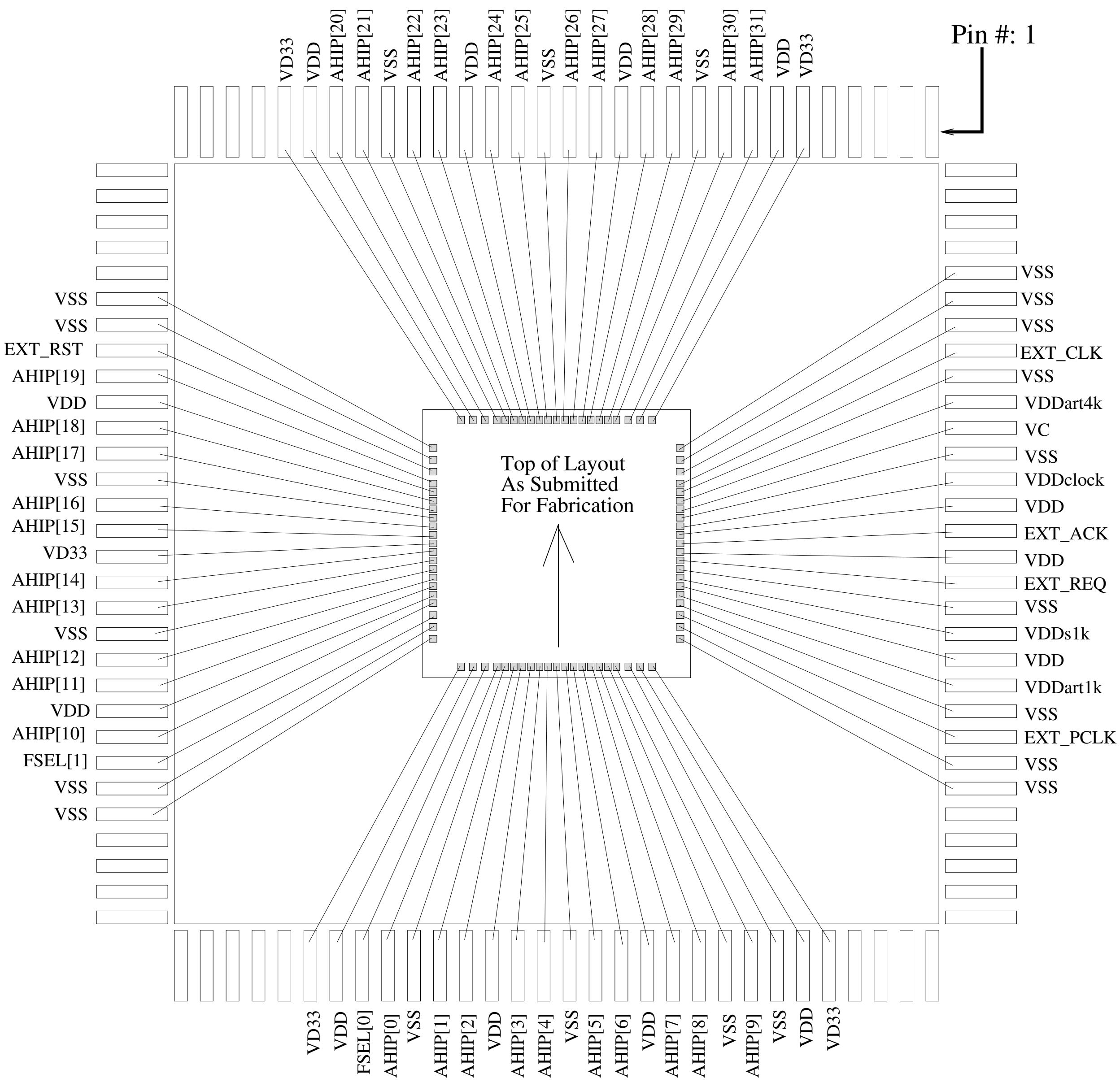


Figure 17: Phyical pinout of ATC0.