QICK PROCESSOR

tProcessor V2

Technical Reference Manual

Table of Contents

1 Introduction

The qick_processor is a custom 32-bit data, 72-bit instruction processor specifically designed for generating timed waveforms, handling data, and triggering events. Figure 1: qick_processor Block Diagram, shows a block diagram of the processor, composed of two main blocks the core, and the dispatcher.

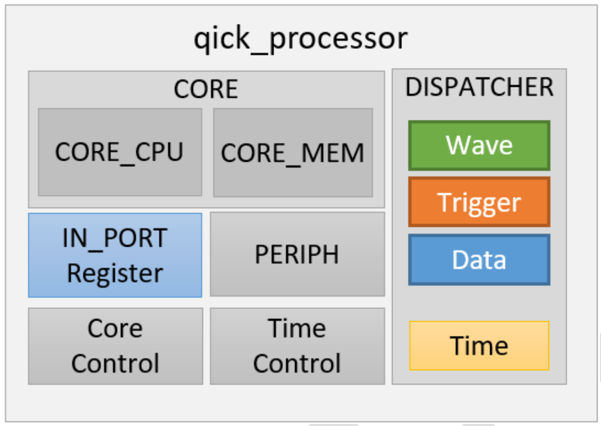


Figure 1: qick_processor Block Diagram

The core serves as the processing unit responsible for executing the program, enabling users to define waveforms and determine output timing. The Dispatcher writes the waveform within the specified time frame.

The qick_processor is compatible with Signal Generators (INT, Mux, and SGV6) for waveform generation and output.

This version is provided with special features including:

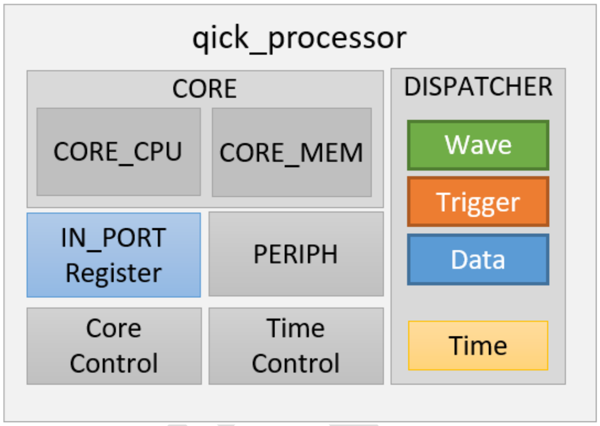
- Multiplication Unit (FPGA DSP): Performs the operation (D±A)*B±C in 2 clock cycles.
- Division Unit (Custom): Provides the quotient and remainder of an integer division in 32 clock cycles.
- Pseudo Random Number Generator (LFSR): Utilizes a Configurable Linear Feedback Shift Register to generate 32-bit pseudorandom numbers.
- Nested CALL functions: Supports nesting up to 256 function calls.
- Debugging Capabilities: Offers step-by-step execution, time stepping, core stepping, status reading, and debug signals.

The qick_processor provides three groups of output ports:

- Four 32-bit Data Ports (DPORT)
- Eight Trigger Output ports (TRIG)
- Sixteen Analog Wave Ports (WPORT)

2 THE QICK_PROCESSOR

The qick_processor is a custom 32-bit data, 72-bit instruction processor specifically designed for generating timed



waveforms, handling data, and triggering events.

Figure 1: qick_processor Block Diagram, shows a block diagram of the processor.

The core (CORE) is composed of 2 main blocks, the Processing unit (CORE_CPU) and the Memory Unit(CORE_MEM). The processing Unit (CORE_CPU) is a 5-stages pipeline Harvard architecture executing one instruction per clock cycle.Each instruction can make a Register, Port, Memory or Branch operation. The processor with 18 instructions, is optimized to execute multiple operations in one instruction. This core version of has no interrupts nor data stack. It has a PC-Stack that enable to nest up to 256 function CALLS. The core also contains a Pseudo Random Number Generator (LFSR), that utilizes a Configurable Linear Feedback Shift Register to generate 32-bit pseudorandom numbers.

The Memory Unit (CORE_MEM) is comprised of a control block and three distinct memory components: the Program Memory (PMEM), a 72-bit memory for storing instructions; the Data Memory (DMEM), a 32-bit memory for user data storage; and the WaveParam Memory (WMEM), a 168-bit memory for storing waveform parameters to be written to the Analog Wave Ports (WPORT)

The Dispatcher (DISPATCHER) is responsible for the timely port signal output. It comprises three FIFOs (Wave, Trigger, and Data), in addition to a set of comparators. Each FIFO holds specific information for output along with designated time. The Dispatcher continuously compares the current time with the scheduled time in the FIFO. Upon reaching the designated time, the Dispatcher updates the Port with the new data.

The AXI stream interface is monitored by a block (DPORT_IN Register) that every time a new data is received, it stores it and update a status bit in the SREG (Special Function Register) s_status.

The qick_processor contains a set of AXI Registers (PROC_xREG) that can be Read and write throu a AXI-Lite interface.

The Advance arithmetic Unit (ARITH) is a FPGA DSP that Performs the operation (D±A)*B±C in 2 clock cycles.

The Division Unit (DIVIDER) is a Custom block that provides the quotient and remainder of an integer division in 32 clock cycles.

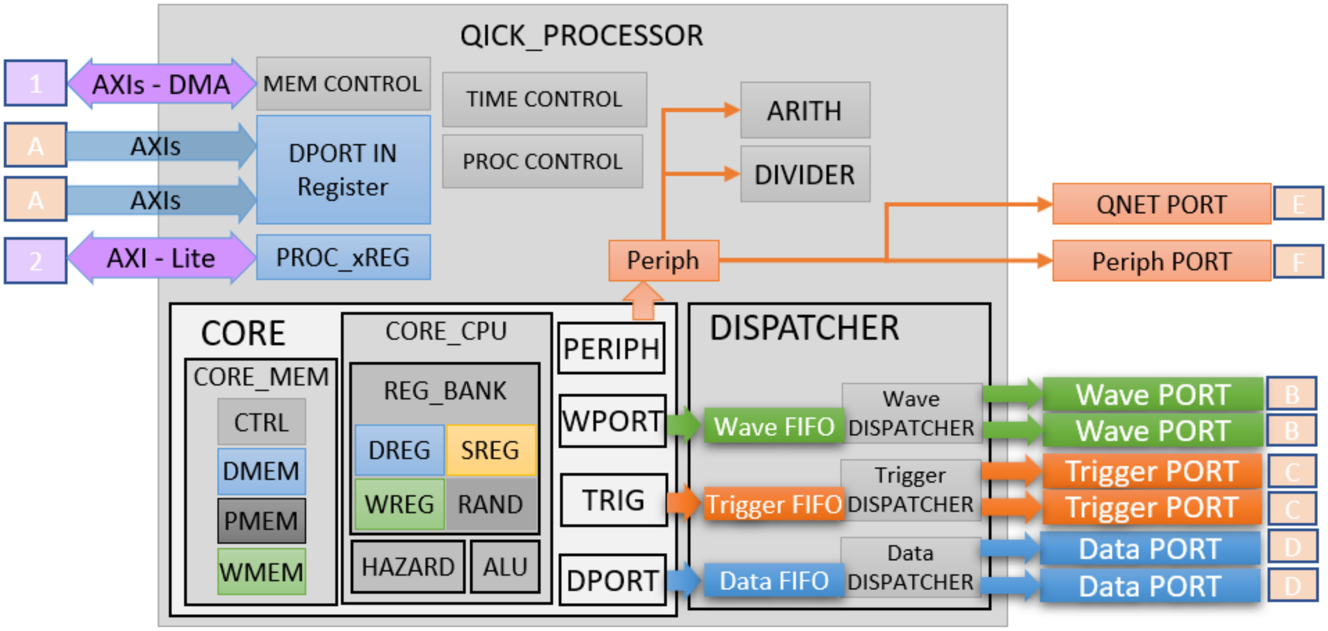


Figure 2: tProcessor-V2 Block Diagram

The output waveform is defined using a 168-bit register for waveform parameters and a 48-bit register for writing time. The WaveParam register consists of six parameters: Frequency (32 bits), Phase (32 bits), Gain (32 bits), Envelope Starting Address (24 bits), Envelope Length (32 bits), and Wave configuration (16 bits).

Users can store sets of WaveParam in the WaveParam memory using the Python interface. The stored waveforms can be accessed and used during program execution. Waveforms can be directly written from the WaveParam Memory to a port or stored in a register for editing before writing to a port.

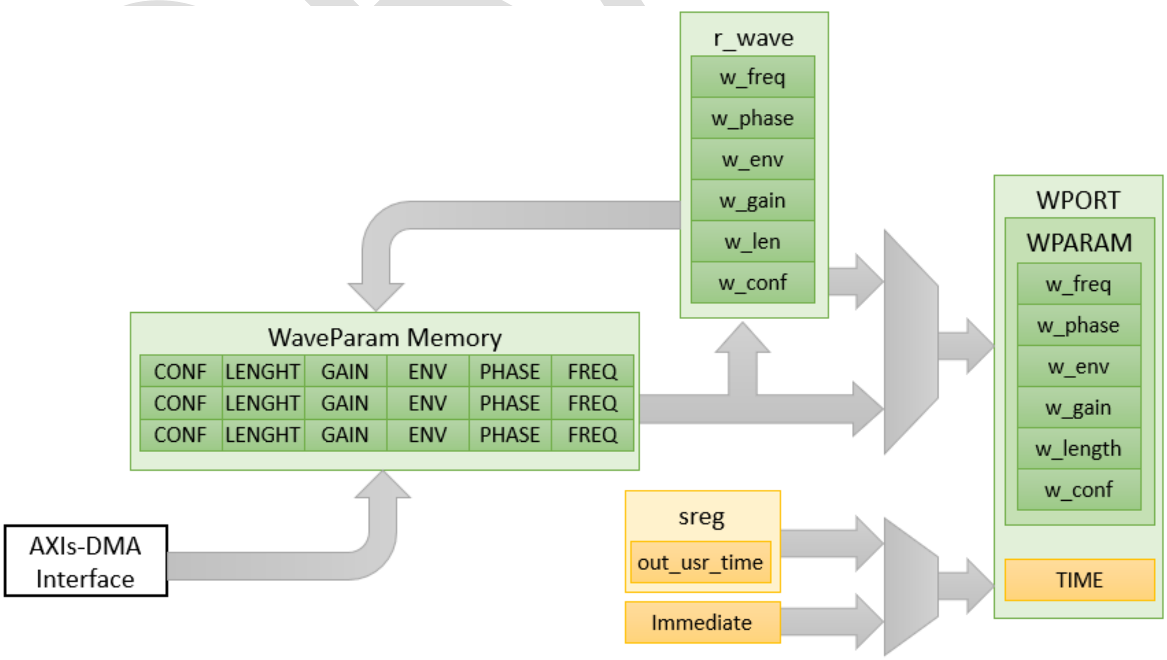


Figure 3: Wave Port Write Sources

The qick Processor has 3 clock Domains

- Core clock (c_clk) : This clock domain belong to the CPU. The peripherals the memories and the Input Port Stram Interface.
- Time clock (t_clk) : This clock domain belong to the Dispatcher. All the Output ports (Trigger, Data and Wave) belong to this domain.
- PS clock (ps_clk) : This clock domain belong to the PS part. Used I AXI Stream DMA and Lite interfaces.

2.1 INTERFACES

The qick_processor provides several interfaces with different functions:

- [1] AXI-Stream DMA Interface to load the Program, Data and WaveParam Memories with a DMA controller
- [2] AXI-Lite Interface to Read / Write AXI Registers.
- [A] Up to Eight AXI-Stream Interfaces for Readout.
- [B] Up to Sixteen AXI-Stream Interfaces to connect SignalGeneratos (Compatible with V6)
- [C] Up to Eight 1-Bits Digital Interfaces to connect Trigger Outpus
- [D] Up to Four 32-Bits Digital Interfaces to connect Digital Data Outpus
- [E] QNET interface for qick network
- [F] Custom Peripheral Port, used to conect a custom made peripheral or co-processor.

2.2 TIMING & DISPATCHER

The time in the qick_processor is measured in clock cycles using a continuously running 48-bit counter. The duration of each clock tick depends on the clock period, which is determined by the configuration of the DAC. For example, with a clock frequency of 256 MHz, the period is 3.9065 ns. If the counter increments by 32 counts, it corresponds to a duration of 125 ns. The maximum time before the counter overflows, assuming a clock frequency of 500 MHz (dependent on DAC configuration), is approximately:

$$2^{48} * 2ns \cong 562.949 \text{ Seconds } \cong \text{ } 9.382 \text{ Minutes } \cong \text{ } 156 \text{ Hours}$$

To facilitate timing operations, the qick_processor employs a 48-bit register called `abs_time` for absolute time measurement. The comparison between the desired output time and the current time is performed using a 48-bit comparator. To obtain the 48-bit representation for the output time, from the 32-bit user time of the processor, a separate 48-bit register called `ref_time` is used as the starting point (t0) for the experiment and to calculate the output timing signals.

The relationship between the different time variables is as follows:

```
    out_abs_time  =  ref_time + out_user_time
current_user_time  =  current_abs_time - ref_time
```

`current_abs_time` : Current value of the absolute time, 48-bit counter that runs at the DAC frequency.
`current_user_time` : It indicates the current time value from the user's perspective.
`out_user_time` : Time set by the user in the instructions and is represented as a 32-bit value.
`out_abs_time` : The 48-bit real-time value at which the output port is written. It is used for comparison with abs_time.
`ref_time` : The reference time serves as the base value or offset used to align the user time to a specific "zero" point.

When `abs_time` reaches the value of `out_abs_time`, the OUT signal is updated on the selected port.

The abs_time starts at 0 and can be reset using an assembler instruction (TIME_RST) or by writing to the AXI-Register **ctrl_reg** a Value 1 from Python. Resetting abs_time also resets the core.

The dispatcher continuously reads the TriggerFIFO, WaveFIFO, and DataFIFO, comparing the output time of the signals with the current abs_time. When the time for writing the signal has already passed (abs_time > out_time), the corresponding wave or data is written to the respective port.

The fastest dispatcher throughput is one data every 5 clock cycles. The dispatcher verifies for time greater only. With a clock of 2.5ns the maximum time for same signal update is 12.5ns.

2.3 AXI Registers

The qick_processor is controlled through the utilization of AXI Registers, employing the AXI interface. Commands from a Python Driver, running in the Processing System (PS) of the FPGA, are used to control, configure, and operate the qick_processor. Refer to Table 1 for a comprehensive list of AXI Registers accessible via the AXI-Python interface.

ADDR	Name	Size	Description	R/W
0	tproc_ctrl	32	Control Commands to qick_processor	R/W
1	tproc_cfg	32	Qick_processot Configuration	
2	mem_addr	16	Starting Address for Memory Operation	R/W
3	mem_len	16	Length of data for Memory Operation	R/W
4	mem_dt_i	32	Data for Single Write Memory Operation	R/W
5	tproc_w_dt1	32	Data to the tProc (Read by the tProc in sreg – s7)	R/W
6	tproc_w_dt2	32	Data to the tProc (Read by the tProc in sreg – s8)	R/W
7	core_cfg	32	LFSR configuration for Core	R/W
8	read_sel	32	Selection of data in the tproc_r_dt register	R/W
9	RFU			
10	mem_dt_o	32	Data for Single Read Memory Operation	RO
11	tproc_r_dt1	32	Data from the tProc (Selected by read_sel)	RO
12	tproc_r_dt2	32	Data from the tProc (Selected by read_sel)	RO
13	time_usr	32	Current_user_time of the tProc (Read by the tProc in sreg - s11)	RO
14	status	32	tProc Status Signals	RO
15	debug	32	tProc Debug Signals	RO

Table 1: AXI Registers

2.3.1 tproc_ctrl

Register **tproc_ctrl** is a 16-bit register, used to control or execute tasks in the qick_processor from the python interface. This AXI-Register **tproc_ctrl** is set to zero automatically once the task was detected. There is no need for the user to reset this register.

AXI-Register	Bit		Description
tproc_ctrl	0	time_rst	Reset the time_abs counter, and the tProcessor. The Instruction pointer return to Zero, all the registers are cleared, and the FIFOs are flushed.
	1	Time_update	Update the time_abs with a specific value. (time_abs = time_abs + time_dt) The core is NOT reset or stopped
	2	Start	Reset the time, reset the core, and start running.
	3	Stop	Stops the execution of the current program. Stops Core and Time.
	4	Core_start	Reset the core unit (The Instruction pointer return to Zero, all the data registers are cleared, and the FIFOs are flushed.) and start running. Time continues running.
	5	Core_stop	Stop the core unit (all registers remain, and FIFOs keep their values) time still running. DATA already in FIFOS will be updated in Ports
	6:12	Debug	Debug Signals (see Debug section)
	13	Flag_set	Set External Flag
	14	Flag_clr	Clear External Flag

Table 2: tproc_ctrl Register Description

2.3.2 tproc_cfg

Register **tproc_cfg** is a 16-bit register, used to configure the behavioral of the qick_processor. It has 2 main functions

- Configure the Memory Controller to Write/Read memories from Python.
- Disable or Enable the external or network control.

AXI-Register	Bit	Name	Description
tproc_cfg	0	START	Start Memory Operation (1-Start) To make a new, Go to 0 first.
	1	OP	Memory Operation selection (0-Read, 1-Write)
	3..2	MEM	Memory Bank Selection for Operation (01-Pmem , 10-Dmem , 11-Wmem)
	4	SRC	Memory Operation Data Source selection (0-AXIS, 1-REGISTERS (Single Read))
	6:5	CORE	Memory Bank Selection (Core0, Core1)
	9	D_QNET	Disable QNET Control (default 0: Yes Control from QNET)
	11	EN_IO	Enable IO Control (default 0: No control from IO)
	10	D_FIFO	Debug (DISABLE FIFO_FULL_PAUSE)

Table 3: tproc_cfg Register Description

2.3.3 core_cfg

Register **core_cfg** is a 8 bit register, is used to configure the behavioral of the LFSR.

CFG	State	Description
00	STOP	Keep the current Number
01	Free Running	Each clock, LFSR Change
10	Change When read	When s1 is read LFSR made a step
11	Change Manually	When s0 is written LFSR made a step

Table 4: LFSR Configuration.

2.3.4 read_sel

Register **read_sel** is a 8 bit register, used to select the source for the **tproc_r_dt1** and **tproc_r_dt2** registers.

READ_SEL	#	TPROC_R_DT
0000	0	TPROC_W_DT
0001	1	CORE0_W_DT
0010	2	CORE1_W_DT
0011	3	DIV
0100	4	ARITH
0101	5	QNET
0110	6	PERIPH
0111	7	PORT
1000	8	CORES_RAND
1001	9	9_9
1010 : 1111	10:15	RFU/DEBUG

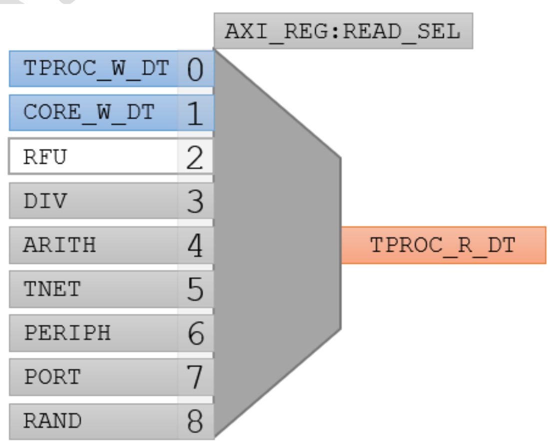


Table 5: read_sel source selection.

2.4 Core-Registers

The qick_processor has 3 register Banks. General Purpouse Register, Special Function Registers and Wave Parameter Registers. Also has Register and a stack for Program Counter, and Flags and Condition registers.

- 32 General purpouse 32-bit registers (***dreg***).
- 16 Special Function Registers (***sreg***)
- Waveform Parameter Register: The 168 bit register is ***r_wave*** each of the 32 or 16 bits registers are ***wreg***

The registers dreg, sreg and wreg can be modified with the REG_WR command

- Program Counter (PC)

The PC is incremnted automatically and can be updateded with the commands JUMP, CALL and RET

- ALU Flag Register (AF)

The ALU FLAGS are updated when the ALU Unit is used and the option -uf is present in the command.

- Internal Flag and external Flag Register (IF, EF)

The internal FLAG can be modified with the assembler instruction `FLAG set` or `FLAG clr`

DREG (32-bits)	
r0	r16
r1	r17
r2	r18
r3	r19
r4	r20
r5	r21
r6	r22
r7	r23
r8	r24
r9	r25
r10	r26
r11	r27
r12	r28
r13	r29
r14	r30
r15	r31

SREG (32-bits)	ASM Name
ZERO	s0 / zero
RAND	s1 / rand
CFG	s2 / s_conf
STATUS	s3 / s_statu[illegible]
DIV_QUOTIENT	s4 / d[illegible]
DIV_REMAINDER	s[illegible]
ARITH_LOW	[illegible]_l
CORE_R_DT1	[illegible]core_r1
CORE_R_DT2	s8 / core_r2
PORT_L[illegible]	s9 / port_l
PO[illegible]	s10 / port_h
CURR[illegible]IME	s11 / tuser
CORE_W_DT1	s12 / core_w1
CORE_W_DT2	s13 / core_w2
OUT USR_TIME	s14 / s_time
PC_JMP_ADDR	s15 / s_addr

SHOULD BE UPDATED

WREG		Bits	ASM Name
r_wave	FREQ	32	w0 / w_freq
	PHASE	32	w1 / w_phase
	ENV	24	w2 / w_env
	GAIN	32	w3 / w_gain
	LENGHT	32	w4 / w_lenght
	CONF	16	w5 / w_conf

Table 6: qick_processor Register Organization

All registers are readable by all instructions. Data for the registers can originate from various sources, including literal (immediate) values, register or ALU outputs, or data retrieved from the Data Memory. Specifically, the 'r_wave' register, a 168-bit register, is generated through the concatenation of all 'wreg' values intended for use within the Wave Bus. This composite register serves the purpose of storage in the WMEM or writing to the WPORT.

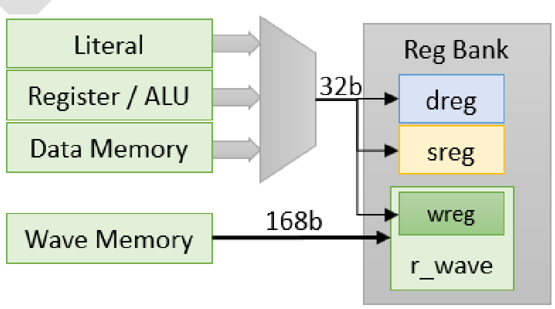


Figure 4: Core Registers Source Data

2.4.1 DREG : Data Registers

The Processor has 32 Data registers of 32 bits. Used as a General Purpose

2.4.2 SREG : Special Function Registers

The Processor has 16 Special Function registers.

- ZERO > Zero value register. In assembler code **`s0`** or **`zero`**
- RAND > 32-bit Pseudorandom number. In assembler **`s1`** or **`rand`**
- CFG > Processor Configuration **`s2`** or **`s_conf`**
- STATUS > Status register. In assembler **`s3`** or **`s_status`**
- DIV_QUOTIENT > 32-bit quotient of Division Unit integer division. In assembler **`s4`** or **`div_q`**
- DIV_REMAINDER > 32-bit remainder of Division Unit integer division. In assembler **`s5`** or **`div_r`**
- ARITH_LOW > Lower 32-bit of Arithmetic Operation. In assembler **`s6`** or **`arith_l`**
- CORE_R_DT1 > Data 1 read from core (data source defined in CFG). In assembler **`s7`** or **`core_r1`**
- CORE_R_DT2 > Data 2 read from core (data source defined in CFG). In assembler **`s8`** or **`core_r2`**
- PORT_LSW > Lower 32-bit of Port Read Operation. In assembler **`s9`** or **`port_l`**
- PORT_MSW > Higher 32-bit of Port Read Operation. In assembler **`s10`** or **`port_h`**
- CURR_USR_TIME > User Time value, In assembler code **`s11`** or **`tuser or curr_usr_time`**
- CORE_W_DT1 > Data 1 from Python to Processor. In assembler **`s12`** or **`core_w1`**
- CORE_W_DT2 > Data 2 from Python to Processor. In assembler **`s13`** or **`core_w2`**
- OUT_TIME > Time register for port writing. In assembler **`s14`** or **`s_time or r_time or out_usr_time`**
- PC_JMP_ADDR > Address register for PC branch. In assembler **`s15`** or **`s_addr or r_addr`**

2.4.2.1 RAND : rand > Pseudo Random Number Register

Special Function Register **rand** (***s1***) is the output of a LFSR. The LFSR is a Fibonacci Serie. The Polynomial implemented is $x^{31} + x^{21} + x^{1} + x^{0}$ which have a maximum-length. A class in python was implemented that calculates the series for simulation. The design was selected based on the Xilinx Application Note XAPP052. Figure 5: LFSR schematic

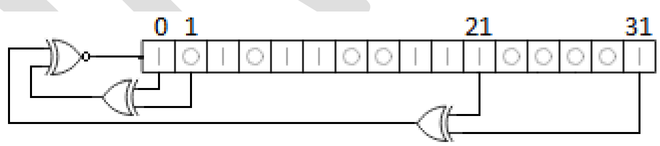


Figure 5: LFSR schematic

The LFSR has 4 working modes, and is configured with the AXI-Register CORE_CFG = **core_cfg[1..0].** Table 4: LFSR Configuration. shows the configuration values for the working modes.

2.4.2.2 *CFG : s_cfg > Core Configuration Register*

The s_cfg register is used to configure the behavior of the CORE. Is a 11 bit register.

SREG	Bit		Description
s_conf	2:0	SRC_DT	Select the source for sreg CORE_R_DT1 and CORE_R_DT1
	5:3	FLAG_SEL	Selection of flag source. For conditional instruction execution -if()
	16	Arith_clr	Clear the signal ***arith_dt_new*** in the sreg ***s_status[0]***
	17	Div_clr	Clear the signal ***div_dt_new*** in the sreg ***s_status[1]***
	18	Tnet_clr	Clear the signal ***tnet_dt_new*** in the sreg ***s_status[2]***
	19	Periph_clr	Clear the signal ***periph_dt_new*** in the sreg ***s_status[3]***
	20	Port_clr	Clear the signal ***port_dt_new*** in the sreg ***s_status[31:16]***

Table 7: sreg s_cfg bit definitions

The lower 16 bitas are used to configure the source od Data and Flags for the Core. The upper 16 bits are used to generate some control actions on the Flags and Status register.

The The sreg CORE_R_DT1 (**s7** or **core_r1)** and CORE_R_DT2 (**s8** or **core_r2**) are register to READ values from different sources.

CORE_REG		
#	SRC_DT	CORE_R_DT1
0	000	TPROC_W_DT
1	001	CORE0_W_DT
2	010	CORE1_W_DT
3	011	TPROC_R_DT
4	100	ARITH
5	101	TNET
6	110	PERIPH
7	111	7

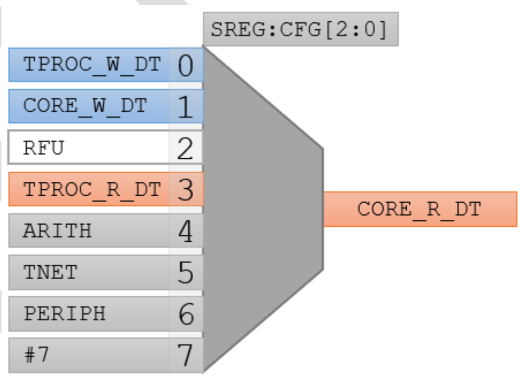


Table 8: sreg s_cfg src_dt bit definitions

FLAG_SEL	Flag Source	Description
00	External_flag	The flag used in the -if(F) r -if(NF) instruction comes from the External flag , set by Python
01	Internal_flag	The flag used in the -if(F) r -if(NF) instruction comes from the Internal flag , set by instruction `FLAG set` or `FLAG clr`
10	port_dt_new	The flag used in the -if(F) r -if(NF) instruction comes from the port_dt_new. This bit is '1' when a NEW data is captured by the IN PORT Register Block. To see the source of the new data see s_status(31:15)
11	qnet_flag	Flag from qnetwork.

Table 9: Flag Selection

Special Constant values are defined in the assembler to make more easy the configuration.

ASSEMBLER EXAMPLES:

```
REG_WR s_conf imm src_tproc          > Select TPROC_W_DT as source for CORE_R_DT
REG_WR s_conf imm src_arith          > Select Arith Data as source for CORE_R_DT
REG_WR s_conf imm src_qnet           > Select QNET Input as source for CORE_R_DT
REG_WR s_conf imm src_periph         > Select Peripheral Input as source for CORE_R_DT
REG_WR s_conf imm flag_int           > Select Internal Flag as source for FLAG condition.
REG_WR s_conf imm flag_axi           > Select AXI Flag as source for FLAG condition.
REG_WR s_conf imm flag_ext           > Select External Flag as source for FLAG condition.
REG_WR s_conf imm flag_div           > Select Division End as source for FLAG condition.
REG_WR s_conf imm flag_arith         > Select Arith End as source for FLAG condition.
REG_WR s_conf imm flag_port          > Select New Port Data as source for FLAG condition.
REG_WR s_conf imm flag_qnet          > Select QNET as source for FLAG.
REG_WR s_conf imm flag_periph        > Select Peripheral as source for FLAG.
REG_WR s_ctrl imm clr_arith          > Clear Arith New data Status bit.
REG_WR s_ctrl imm clr_div            > Clear Division New data Status bit.
REG_WR s_ctrl imm clr_qnet           > Clear QNET New data Status bit.
REG_WR s_ctrl imm clr_periph         > Clear PERIPH New data Status bit.
REG_WR s_ctrl imm clr_port           > Clear Data Port IN New data Status bit.
DIV r1 r2                            > Make r1 / r2
REG_WR s_conf imm flag_div           > Select Division End as source for FLAG condition.
JUMP HERE -if(NF)                    > Wait until Division ends.
```

2.4.2.3 STATUS: s_status > Status Register

The Status register **s_status** is used to get the status of the peripherals, the Data and the FIFO state.

It indicates when a peripheral is ready to use, or if new data was new data arrives to the In Port or if a new data

SREG	Bit	Name	Descrition
s_status	0	ARITH_DT_NEW	New data is present in the ARITH Peripheral
	1	DIV_DT_NEW	New data is present in the DIVIDER Peripheral
	2	TNET_DT_NEW	New data is present in the TNET Peripheral
	3	PERIPH_DT_NEW	New data is present in the CUSTOM EXTERNAL Peripheral
	4	ARITH_RDY	The Peripheral ARITH is ready to use
	5	DIV_RDY	The Peripheral DIVIDER is ready to use
	6	TNET_RDY	The Peripheral QNET is ready to use
	7	PERIPH_RDY	The CUSTOM EXTERNAL Peripheral is ready to use
	8	DFIFO FULL	The Data FIFO is FULL
	9	DFIFO EMPTY	The Data FIFO is EMPTU
	10	WFIFO FULL	The Wave FIFO is FULL
	11	WFIFO EMPTY	The Wave FIFO is EMPTY
	16:31	Port_dt_new	16 bit array indicating if new data was received in the IN_PORT

Table 10: sreg s_status bit definition

2.4.3 WREG : Wave Param Registers

The Processor has 6 registers used to define the parameters of a Waveform.

- w_freq : Is a 32 bit register to define the Frequency of the WaveForm
- w_phase : Is a 32 bit register to define the Phase of the WaveForm
- w_env : Is a 24 bit register to define the Starting Address of the Envelope for the WaveForm
- w_gain : Is a 32 bit register to define the Starting Address of the Envelope for the WaveForm
- w_lenght : Is a 32 bit register to define the Length of the Envelope for the WaveForm
- w_conf : Is a 16 bit register to Configure the options of the WaveForm

Single parameter register (wreg) can be accessed for register manipulation with the instruction REG_WR or the 168bit Complete waveform (r_wave) can be accessed to WMEM or WPORT write/read with the instructions WMEM_WR or WPORT_WR.

2.5 CORE -MEMORY

The processor has 3 Memories, with different interfaces for Program, Data, and Waveform to enable simultaneously instruction fetch and data load/store.

- Program Memory (*PMEM*): Stores the program to be executed. Is a dual port 72-bit memory (Optimized for FPGA BRAM), accesible from the qick_processor and from Python interface.
- Data Memory (*DMEM*): Stores 32-bit user data. Is a dual port 32-bit memory, accesible from the qick_processor and from Python interface.
- WaveParam Memory (*WMEM*): Stores the parameters needed to define a waveform to be written in the WPORT. The parameters are Frequency, Phase, Gain, Envelope, Length, and configuration. Is a dual port 168-bit memory, accesible from the qick_processor and from Python interface.

The qick_processor has a separate Memory address Calculator for PMEM, DMEM and WMEM, leaving the ALU free to operate, and being able to store results from the ALU in the Data Memory.

Read (and Write) from (to) memories from the PS interface can be done in two different ways. Using a 256-bit DMA controller or using the AXI-Register Interface. The AXI-Register *TPROC_CFG* is used to config the process. See Section

Python Interface (pag 44) for instruction in how to read / write the memories.

2.5.1 PMEM > Program Memory

This memory stores the program to be executed. Is a configurable memory from 256 Spaces to 65536. Each instruction is a 72bit word. The Program Counter (PC) stores the address of the current instruction. The address for the next instruction, depending on the current instruction, can be the PC+1(no branching instruction) , a Literal (branching instruction) Value, the value of the sreg s_addr (Branching instruction) or the value fom the PC_stack (RET instruction) . The sreg **s_addr** is the ONLY register used to jump.

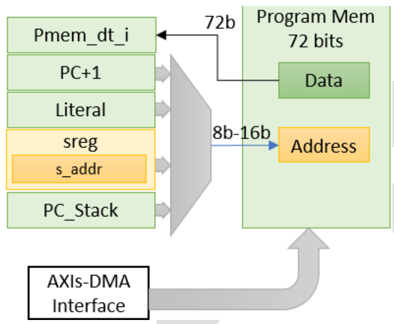


Figure 6: PMEM Access

2.5.2 DMEM > Data Memory

This memory store general purpose 32bit data. Is a configurable memory from 256 to 65536 Words.

Addressing Modes

- The Processor has 4 addressing modes for the Data Memory address
  - Literal > The address is an Immediate Value
  - Register > The address is stored in a Register
  - Indexed Literal > The address is the Sum of a Register Value and an Immediate Value
  - Indexed Register > The address is the Sum of 2 registers.

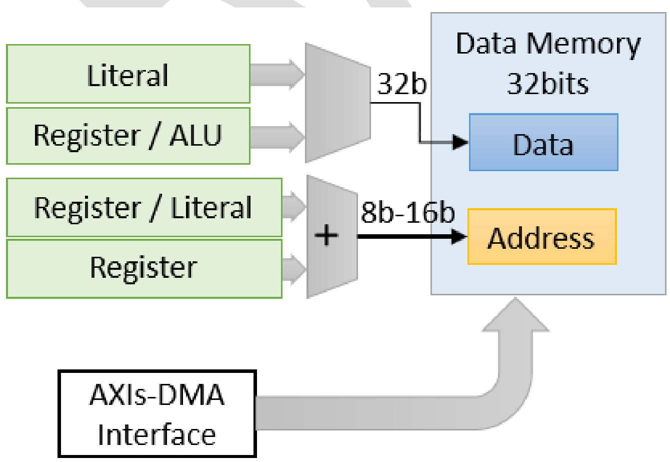


Figure 7: DMEM Access

2.5.3 WMEM > Wave Param Memory

This memory store Wave Parameters. Is a configurable memory from 256 to 2048 Spaces connected to the Wave Bus, this is a 168bit data bus.

- The WaveParam Memory address has 2 addressing modes

- Literal > The address is an Immediate Value
- Register > The address is stored in a Register

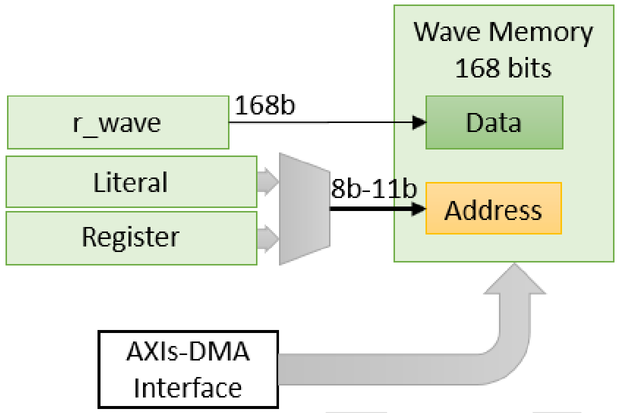


Figure 8: WMEM Access

2.6 Qick Ports

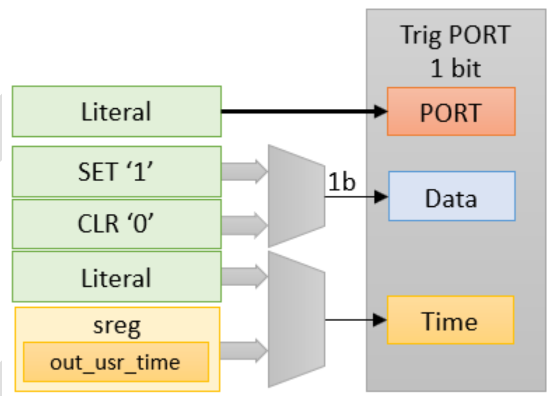
The qick_processor provides three groups of output ports and one Input port.

- Four 32-bit Output Data Ports (DPORT)
- Eight Trigger Output ports (TRIG)
- Sixteen Analog Wave Output Ports (WPORT)
- Sixteen Data Input Ports (IN_PORT)

2.6.1 Output Ports

2.6.1.1 TRIG

This is a Single BIT output. Intended to generate external single bit trigger signals. Output Time can be selected from a Literal (Immediate) value or the sreg ***out_usr_time***.



Assembler instruction:

```
TRIG <Set/Clear> <Port>
```

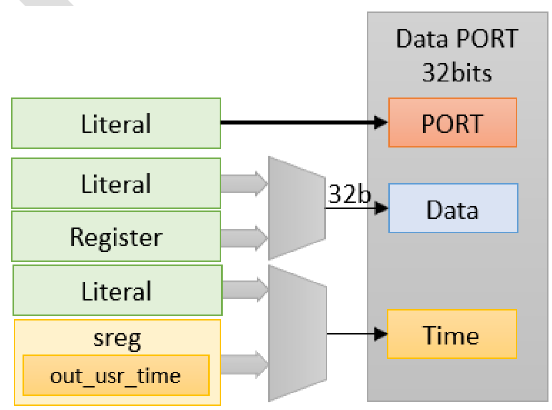
ASSEMBLER EXAMPLES:

```
TRIG p0 set @150
TRIG p1 clr -wr(r1 imm) #2
TRIG p2 set
```

2.6.1.2 DPORT

This is a 1-bit to 32-bit configurable output. Is designed to send out digital information at specific time.

The data for this port can be sourced from either a Register or a Literal (Immediate) value. Output Time can be selected from a Literal (Immediate) value or the sreg ***out_usr_time***.



Assembler instruction:

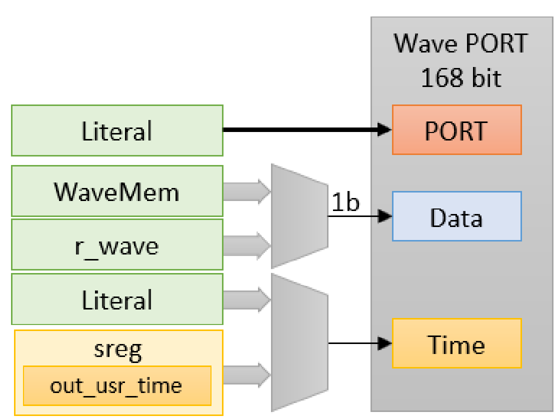
```
DPORT_WR <Port> <Source> <Time>
```

ASSEMBLER EXAMPLES:

```
DPORT_WR p0 imm 1 @125
DPORT_WR p0 reg r3
```

2.6.1.3 WPORT

The 168-bit WaveParam output is designed for connection to the qick_sg_translator block, which can manage three different signal generators: sig_gen_v6, sg_mux, and sg_int. The data for this port can be sourced from either the r_wave register or the WMEM memory. Output Time can be selected from a Literal (Immediate) value or the sreg ***out_usr_time***.



Assembler instruction:

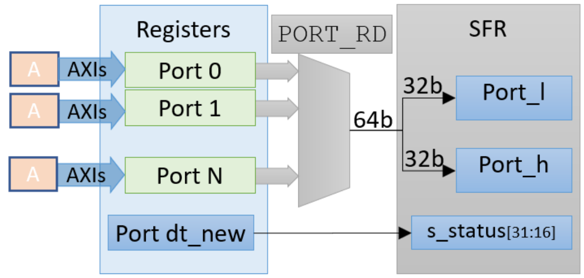
```
WPORT_WR <Port> <Source> <Time>
```

ASSEMBLER EXAMPLES:

```
WPORT_WR p0 r_wave
WPORT_WR p1 r_wave @125
WPORT_WR p2 wmem [&2]
```

2.6.2 Input Ports

This is a 1 to 16 configurable input port array consisting of 64-bit (two 32-bit registers) with an AXI Stream interface. Upon receiving data (with t_valid asserted in the AXI interface), the data is stored in a register, simultaneously setting a bit to indicate the arrival of new data in the sreg ***s_status***. The DPORT_RD instruction copy the selected registered data, belonging to a port, to the Special Function Registers ***port_l*** and ***port_h***.



Assembler instruction:

```
DPORT_RD <Port>
```

ASSEMBLER EXAMPLES:

```
DPORT_RD p0
DPORT_RD p1
```

2.7 Peripherals

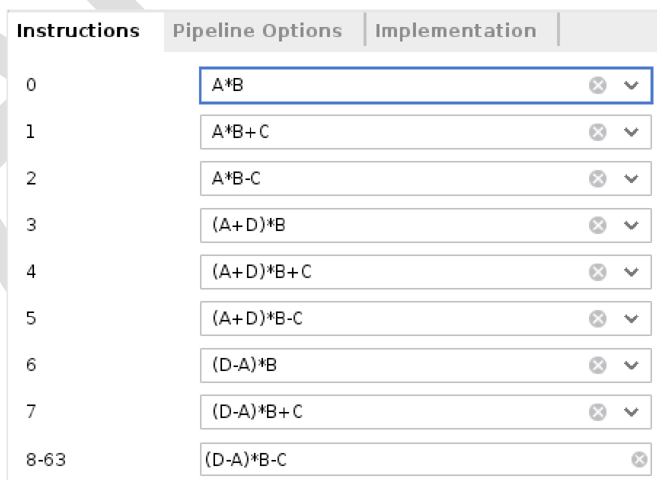
The qick_processor is equipped with two internal peripherals, one used for multiplication and the other for division. Additionally, the processor provides connections for two external peripherals, intended for use with the qick_network block and a custom peripheral.

2.7.1 Advanced Arithmetic Unit

This block is an instance of the FPGA DSP.

It can perform 9 different variants of the operation (D ± A) * B ± C in 2 clock cycles.

The 64-bit result value is stored in Special Function Registers **arith_l**(**s2**) and **arith_h**(**s3**).



Assembler instruction:

```
ARITH <Option> <Sources>
```

ASSEMBLER EXAMPLES:

```
ARITH T w_freq r1
ARITH PT r1 r2 r3
```

2.7.2 Division Unit

The Division Unit (DIVIDER) is a custom block designed to compute the quotient and remainder of an integer division within 32 clock cycles. It operates with 32-bit inputs for the numerator and the denominator, and it produces 32-bit output for the quotient and the remainder.

The numerator is always a register, the denominator can be a register or a 24-bit literal (Immediate) value. The two 32-bit result values are stored in Special Function Registers **div_quotient** (**s4**) and **div_remainder** (**s5**).

Assembler instruction:

```
DIV <Num> <Den>
```

ASSEMBLER EXAMPLES:

```
DIV r1 r2
DIV r1 #100
```

2.7.3 External Peripheral

The qick_processor is equipped with an output interface designed to connect to a custom peripheral. This interface is composed of the next signals:

- qp_en (Enable): Activates the peripheral on the rising edge.
- qp_op (5-bit Operation): A 5-bit word specifying the operation to be performed by the peripheral.
- qp_dt_o (Four 32-bit Data): These ports transmit 32-bit data from the qick processor to the peripheral.
- qp_rdy (Ready): Indicates the current state of the peripheral. Connected to the status register (s_status).
- qp_vld (Valid): Marks the new data, indicating the availability of valid data to the qick_processor.
- qp_dt_i (Two 32-bit Data): These signals are the data outputs of the peripheral, to the qick_processor.
- qp_flag (Flag): Although its specific purpose is not explicitly stated, it appears to be a flag related to the peripheral's operation.

On the rising edge of the 'Enable' signal, the Peripheral should capture both the Data and the Operation, subsequently deactivating the ‘qp_rdy' signal to let know that the device is busy.

After completing the operation, the peripheral should write the result into the ‘qp_dt' together with the ‘qp_vld’ port and raise the ‘qp_rdy' signal.

The signal ‘qp_vld’ is used by the qick_processor to register the input and to set the q*p_dt_new* bit on the ***s_status[9]*** for QPA and ***s_status[11]*** for QPB. The signal ‘qp_rdy’ is connected to ***s_status[8]*** for QPA and ***s_status[10]*** for QPB. ***.***

The assembler software can use the ‘q*p_dt_new’* bit on the ***s_status[9, 11]*** or the ‘qp_rdy’ bit in ***s_status[8, 8]*** to check for the finalization of the task. Peripheral can generate a Flag and it can be used as a condition for processing.

To send command to the Peripheral the command `PA` or `PB` is used.

To read the data from the peripheral the QPA or QPB source should be selected using ***s_conf[3:0]*** and then read from the sreg **core_r1**(**s7**) and **core_r2**(**s8**).

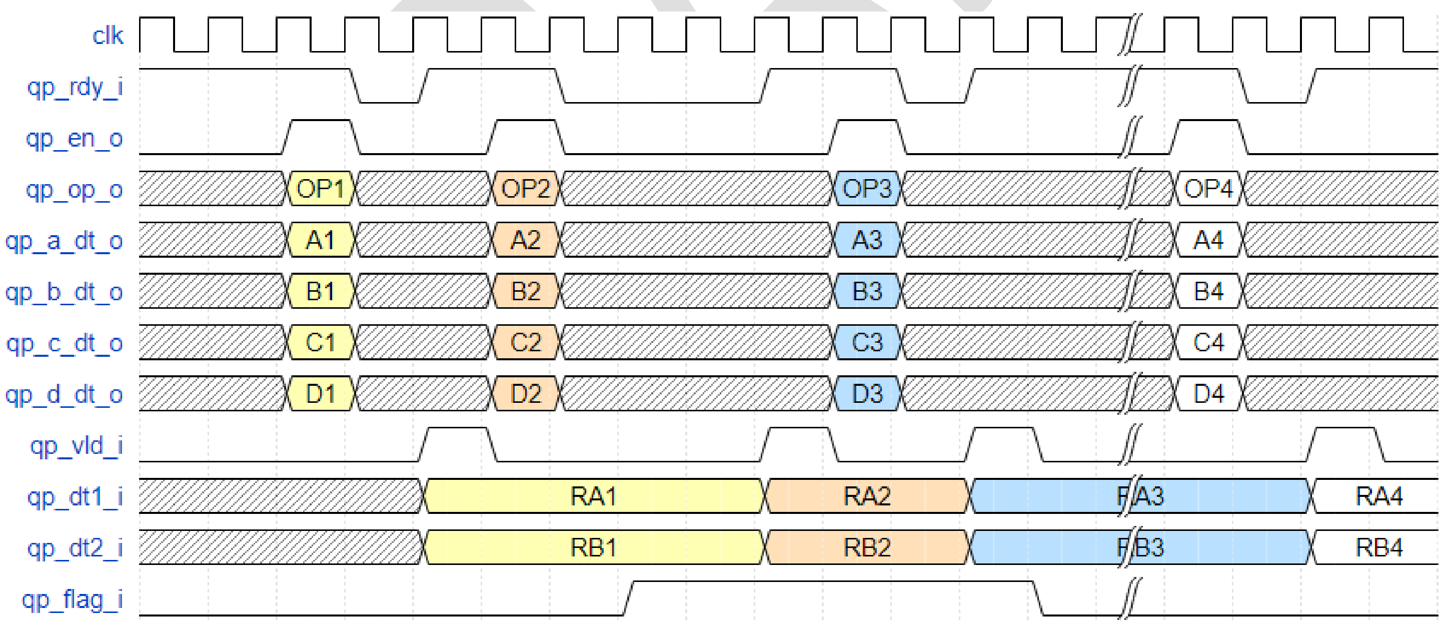


Figure 9: Time Diagram for Custom Peripheral

Assembler instruction:

```
PA <OP> <DTA> <DTB> <DTC> <DTD>
```

ASSEMBLER EXAMPLES:

```
TEST -op(s_status AND qpa_rdy)       > Check for QPA READY
JUMP PREV -if(NZ)
```

PA 31 r1 r2 r3 r4 REG_WR s_conf imm qpa_src
REG_WR s_conf imm qpa_flag

2.8 CONTROL THE QICK_PROCESSOR

The qick_processor has 2 control state machines. The fists one “core_st”, in c_clk (the CORE clock) domain, manages the state of the core. The second one “time_st”, in t_clk (DISPATCHER clock) domain, controls the qick processor time used by the dispatcher.

Control commands

- ***Start*** > Starts the qick_processor. Reset the time, reset the core and start running.
- ***Stop*** > Starts the qick_processor Stops the execution of the current program and the time.
- ***Time Reset*** > Reset the `time_abs` counter and the core. The Instruction pointer return to Zero, all the registers are cleared, and the FIFOs are flushed. Once the core is reset, if it was running goes to running state but if it was stopped, after the reset it goes to stop state.
- ***Time Init*** > Initialize the `time_abs` with a specific value. The core is NOT reset or stopped; it keeps running.
- ***Time Update*** > Increment the `time_abs` with a specific value. (time_abs = time_abs + time_dt) The core is NOT reset or stopped.
- ***Core Start*** > Reset the core unit (The Instruction pointer return to Zero, all the data registers are cleared, and the FIFOs are flushed.) and start running. Time is not interrupted, continues running.
- ***Core Stop*** > Stop the core unit (all registers remain, and FIFOs keep their values) time still running. WRITE PORT already in FIFOS will be executed.

Some controls can be executed from IO inputs, from the QNET interface from python and from the CORE. Table 11 : qick processor control shows all the actions that can be done by the IO, the python interface, the QNET interface and the Core.

- The External IO pins. The qick processor has 2 external inputs proc_start and proc_stop that start and stop the qick_processor.
- QNET Network Commands. Time and Core can be controlled trough the network. The commands can be generated with assembler instructions running in the core or with a Python interface with the QNET block.
- PYTHON running in the PS trough the AXI interface (with the tproc_ctrl AXI Register).
- CORE instructions running in the PMEM. Time can be rese and update with assembler instructions.

CMD	IO	QNET	PYTHON	CORE	Time		Core	
start	Y		Y		RESET	RUN	RESET	RUN
stop	Y		Y			STOP		STOP
reset			Y		RESET	STOP	RESET	STOP
run			Y			RUN		RUN
time_reset		Y	Y	Y	RESET	RUN	RESET	PREVIOUS
time_init		Y			INIT	RUN		No Change
time_update		Y	Y	Y	UPDATE	RUN		No Change
core_start		Y	Y			No Change	RESET	RUN
core_stop		Y	y			No Change		STOP

Table 11 : qick processor control

The qick_processor has an internal (IF) and an external (EF) flag resister used as a condition for instruction execution. The external flag can be set and clear with python commans

- ***Flag set*** > Set the External Flag (EF) .
- ***Flag clear*** > Clear the External Flag (EF).

Debug commands and status are explained in Section Debugging Pag(53)

3 THE SIGNAL GENERATOR

The 168-bit WaveParam output is designed for connection to the qick_sg_translator block, which can manage three different signal generators: sig_gen_v6, sg_mux, and sg_int This section describes briefly the sig_gen_v6.

The SG generates a waveform from 4 posible sources. Table, DDS, Product (Table * DDS) and Zero-value. The Table is a custom waveform with an arbitrary shape. The DDS is a complex cosine/sine generator block, whose frequency can be configured using the provided interface.

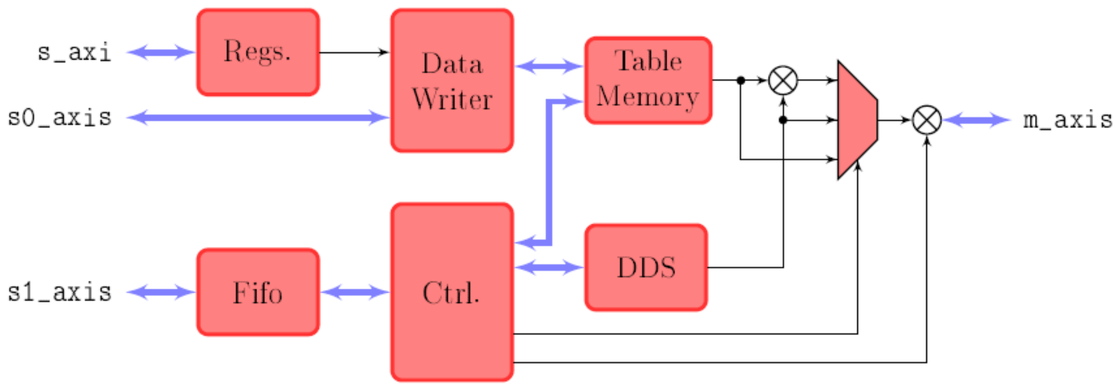


Figure 10: Signal Generator Block Diagram

The table is loaded using the AXI-Stream interface with a support DMA block. User can specify the address of the first sample using the corresponding axi register, to allow uploading several waveforms into the internal memory. The DDS section is an integrated IP that works in "streaming" mode, which means frequency can be changed from sample to sample. This allows the user to specify a precise duration for waveforms. The number of samples of the output waveform is specified in the input interface. This configuration interface allows to push waveforms into the internal queue (FIFO). Whenever the FIFO is empty,the block will output zero-valued samples. When the FIFO is not empty, the block will act accordingly to generate the waveform at the output m_axis interface.

4 INSTRUCTION SET

The qick_processor has 18 instructions and the capability to perform multiple tasks in the same instruction.

4.1 SUMMARY

Operation	Assembler
No Operation	NOP
Test Register Value with ALU operation and update Flags.	TEST <op>
Write Register (Rd: Data Register, Special Register or Wave Parameter Register) from Source (Sources : ALU Operation, Data Memory, WaveParam Memory or Immediate value)	REG_WR <Rd> <Source>
Write data to Data Memory.	DMEM_WR [Address] <Source>
Write Wave Memory. Copy the **r_wave** register value to the specified address in the Wave Memory.	WMEM_WR [Address}
Set or clear TRIGGER port	TRIG <Set/Clear> <Port>
Write a 32-Bit Data to specific Data Port at Specific time (Time is always the **r_time** register Value)	DPORT_WR <Port> <Source> <Time>
Update the value of the Special Function Registers 8 and 9 to the value of the selected port.	DPORT_RD <Port>
Write a Wave to specific Wave Port at Specific time (Time can be an Immediate Value or a registered Value)	WPORT_WR <Port> <Source> <Time>
Conditional JUMP to a specific Address (Address can be a Label, an Immediate Value or a Register)	JUMP [Address]
Function CALL to a specific Address. CALL can be Nested Up to 4 times	CALL [Address]
RETURN from current CALL to previous PC.	RET
Set or Clears internal FLAG value.	FLAG
Command to control time_ref and time_abs in Processor. Reset Absolute Time (time_abs = 0), Change Time Ref (time_ref = Imm_value), Increase Time Ref (time_ref = time_ref + Imm_value)	TIME <Option> <Value>
Make (D ± A) * B ± C Arithmetic operation in 2 clock cycles. 64-bit result value is stored in Special Function Registers 2 and 3	ARITH <Option> <Sources>
Calculate Quotient and the Reminder of the division. It takes 32 clock cycles. Is done by a co-processor	DIV <Num> <Den>
Commands to the QNET periphery	NET <cmd>
Execute Custom Peripheral Instruction	PA/PB OP A B C D
Wait until User Time arrives to specific value.	WAIT @Time

4.2 INSTRUCTIONS OPTIONS

All instructions have optional arguments to configure and perform different tasks. The user can specify to execute a second instruction in the same Scan execute a second optional instruction in the same command. There are Second Data Wave and Port Write instructions also a TEST can be done.

Description	Option
ALU Operation	`-op(<Operation>)`
Update Flags	`-uf`
Conditional Command Execution	`-if(<Condition>)`
Specify the Time with an Immediate Value	`@<Time>`
Dual Task, Add a Write Register Task to the command	`-wr(<Dest> <Source>)`
Dual Task, Add a Write Port Task to the command	`-wp(<Port>)`
Dual Task, Add a Write WaveMemory Task to the command	`-ww`

Table 12: Assembler Instruction Options

ASSEMBLER EXAMPLES:

REG_WR r0 op -op(r1-r2)	> Stores in ***r0*** = ***r1-r2***
REG_WR r0 op -op(r1-r2) -uf	> Stores in ***r0*** = **r1-r2** and update ***S*** and ***Z*** Flags
JUMP [LABEL] -if(NZ)	> Jump to LABEL Address if Flag is NonZero
WPORT_WR r_wave p1 @100	> Write ***r_wave*** to Wave Port 1 at Time 100

4.2.1 ALU Operation < -op >

The Arithmetic & Logic unit can perform 15 different operations. When executing instruction REG_WR all 15 operations are available, but when executing ALU operations as a Second option, only 4 are available. Table 13: ALU Operations shows all operations and in gray the 4 for SDT.

Code				Operation	
0	0	0	0	ADD	Addition
0	0	0	1	SUB	Subtraction
0	0	1	0	AND	Logical bitwise AND
0	0	1	1	ASR	Arithmetic Shift Right
0	1	0	0	ABS	Absolute Value
0	1	0	1	MSH	Most Significative Half (16MS Bits)
0	1	1	0	LSH	Least Significative Half (16LS Bits)
0	1	1	1	SWP	Swap Half Word (16LSB-16MSB)
1	0	0	0	NOT	Logical bitwise NOT
1	0	0	1	OR	Logical bitwise OR
1	0	1	0	XOR	Logical bitwise XOR
1	0	1	1	CAT	Concatenate 2 LSB 16Bits
1	1	0	0	RFU	Reserved Future Use
1	1	0	1	PAR	Parity
1	1	1	0	SL	Logic Shift Left
1	1	1	1	SR	Logic Shift Right

Table 13: ALU Operations

Operation	Description	Assembler Code
NONE	Copy the value of a Register	`-op(r0)`
ADD	Register plus Immediate value	`-op(r1 + #7)`
	Register Plus Register value	`-op(s1 + r2)`
SUB	Register minus Immediate value	`-op(r1 - #7)`
	Register minus Register value	`-op(s1 - r2)`
ASR	Arithmetic Shift Right Immediate Amount (MSB completed with Sign bit, up to 15 shifts)	`-op(r1 ASR #7)`
	Arithmetic Shift Right Immediate Amount	`-op(s1 ASR r2) *(r2 < 16)`
ABS	Absolute Value	`-op(ABS r1)`
MSH	Most Significan HalfWord	`-op(MSH r2)`
LSH	Least Significan HalfWord	`-op(LSH r3)`
SWP	Swap HalfWord	`-op(SPW r4)`
CAT	Concatenate 2 HalfWord 16 Bits	`-op(r5 CAT r6)`
PAR	Parity Check	`-op(PAR r7)`
SL	Shift Left (from 0 to 15 shifts)	`-op(r8 SL r3)`
SR	Shift Right (from 0 to 15 shifts)	`-op(r9 SR r3)`
NOT	NOT Register	`-op(NOT r1)`
AND	Register AND Immediate	`-op(r1 AND #7)`
	Register AND Register	`-op(s1 AND r2)`
OR	Register OR Immediate	`-op(r2 OR #7)`
	Register OR Register	`-op(r2 OR r2)`
XOR	Register XOR Immediate	`-op(r3 XOR #7)`
	Register XOR Register	`-op(r3 XOR r2)`

Table 14: ALU Operations Examples

Is not possible to do a literal minus a rgister, only a rgister minus literal.

4.2.2 Update Flag < -uf >

This option updates the value of the ALU flas (S and Z) with the current operation value.

Every time the ALU is used, it generates the Sign and Zero Flag, but the flag is not stored unless it is explicated in the -uf option.

4.2.3 Conditional Execution < -if() >

All Data movement instructions contains a condition field which determines whether the CPU will execute them. This removes the need for many branches, which stall the pipeline (2 cycles to refill) This Allows very dense in-line code, without branches (but more memory is used). User must decide if the time penalty of not executing several condition instructions has more or less overhead of the branch.

The Condition Field is a 3 Bits field, what gives up to 7 conditions. There are 4 ALU related Condition, two Time related and two FLAG related conditions. To use the condition in the Assembler just add the -if() option

By default, data processing operations do not affect condition flags. To cause the condition flag to be updated, the -uf (update Flag) option should be included in the instruction, as explained in 4.2.2.

Table 15: Conditions shows a list of the 15 conditions.

	CODE			Condition
	0	0	0	ALWAYS
Z	0	0	1	Zero
S	0	1	0	Sign (Negative)
NZ	0	1	1	Not Zero
NS	1	0	0	Not Sign (Positive)
F	1	0	1	FLAG
NF	1	1	0	NO FLAG
	1	1	1	RFU

Table 15: Conditions

The FLAG for -if(F) and -if(NF) depends on the value of s_cfg register (see 0).

The condition applies for the write or jump, and -uf flag

REG_WR r0 imm #4 -op(r4-#4) -uf -if(Z)

REG_WR r0 imm #5 -op(r5-#5) -uf -if(Z)

REG_WR r0 imm #6 -op(r6-#6) -uf -if(Z)

If the condition is not fill, the write is not done, and the flag update is not update

If the instruction is not executed neither the -wr option
JUMP ERROR -if(NZ) -wr(r0 imm) -op(r2-#2) -uf
JUMP ERROR -if(NZ) -wr(r0 imm) -op(r3-#3) -uf

ASSEMBLER EXAMPLE:

`REG_WR r0 imm #0`	`> r0 = 0`
`REG_WR r0 op -op(r1+r2) -if(Z)`	`> r0 = r1+r2 IF Z flag is set`
`JUMP [LABEL] -if(Z)`	`> Jump to LABEL Address if Flag is NonZero`

The External Condition can be set and clear by the Tproc (with the COND command), By Python (Writing in the TPROC_CTRL Reg bit 10 SET bit 11 Clear), is used to control the status of the divider and also is set to one if a New Data is captured in some of the Input PORTS.

The External Condition can be set by> tProc / Python / Port / Divider
The External Condition can be clear by> tProc / Python / Divider

ASSEMBLER EXAMPLE:

`REG_WR r0 op -op(r2-#10)`	`> r0 = r2-10 no Flag Update`
`REG_WR r0 op -op(r2-#10) -uf`	`> r0 = r2-10 and SET FLAGS`
`COND set`	`> Set External Condition Flag`
`COND clear`	`Clear External Condition Flag`
`DIV r10 r11`	`> Clear External Condition Flag makes the Division (r0 / r11) and set the FLAG when finish.`
`TIME set_cmp r4`	`> Set the Time Condition value to the value stored in register 4.`

4.2.4 Dual Task Instructions. < -wr(), -wp(), -ww >

Depending on the options used, the instruction can also execute a second optional task in the same command. There are Second Data (-wr()) Wave (-ww) and Port (-wp()) Tasks also a TEST (-op() -uf) can be done.

Command	WR Data Task	WW Wave Task	WP Port Task	Conditional Execution	Update Flag
NOP					
TEST					YES
REG_WR				YES	YES
REG_WR	*YES	YES	YES		YES
DMEM_WR	YES			YES	YES
WMEM_WR	YES	NA	YES		YES
TRIG					
DPORT_WR	YES	YES		YES	YES
DPORT_RD					
WPORT_WR	YES	YES	NA		
JUMP	YES		YES	YES	YES
CALL	YES				YES
RET	YES				YES
FLAG					
TIME	YES		YES		YES
ARITH					
DIV					
COND					

*When REG_WR source is dmem, a TEST can be done to update the flags.
Second Data Instruction can be:

- Register Write: To enable second register write, the option -wr(<Rd> <Source>) should be added to the command.
- Register Test: To enable a register test, the operation -op() and the -uf option should be added.

hexa

4.2.4.1 Write Register

The write register option is added with the -wr(dest, source) it has 2 operands destination register and source. Destination register can be a (***dreg***, ***sreg*** or **wreg**) register. Source can be *imm* or *op.* Only one immediate (Literal) value can be used per instruction and up to 2 register values. As shown in Table 18: Data Source Format, if no register value is used, Immediate value is a 32-bit value. If a register is used, Immediate value is a 24-bit value if 2 registers are used the immediate value is a 16 bit value.

When using SDT (Second Data Task)

Operation	Description	Example
ADD	Register plus Immediate	`-op(r1 + #7)`
	Register Plus Register	`-op(s1 + r2)`
SUB	Register minus Immediate	`-op(r1 - #7)`
	Register minus Register	`-op(s1 - r2)`

AND	Arithmetic Shift Right Immediate Amount	`-op(r1 ASR #7)`
	Arithmetic Shift Right registered Amount	`-op(s1 ASR r2)`
ASR	Arithmetic Shift Right Immediate Amount	`-op(r1 ASR #7)`
	Arithmetic Shift Right registered Amount	`-op(s1 ASR r2)`

ASSEMBLER EXAMPLES:

`-wr(r0 imm) #5`	> SDT: Write the value 5 in ***r0***
`JUMP [LABEL] -wr(r1 op) -op(rand)`	> SDT: Copy the random value in ***r1***
`DMEM_WR [r0+r1] imm #5 -wr(r4 op) -op(r2+r3)`	> Write to Data Memory Address [***r0-r1]*** the literal value 5 AND Write ***r4*** the operation ***r2+r3***
`REG_WR`	

4.2.4.1 Write Wave

4.2.4.2 Write port

The write port option is added with the -wp (source) it has 1 operand, the source of the data. Source can be r_wave or mem*. The time used for SPT is the one in r_time.*

ASSEMBLER EXAMPLES:

`REG_WR r_wave [&3] wp(r_wave) -p3`	> Write the operation ***r0***+1 to Data Port 0 AND to ***r0***
`WMEM_WR [&3] -wr(r_gain op) -op(r_gain+#25) -wp(r_wave) -p3`	> Write to Data Memory Address 3 the operation ***r1-r2*** AND Write ***r3*** the Immediate Value 1
`REG_WR r_wave wmem [&3] -wr(r5 op) -op(r6-#1) -uf -wp(r_wave) -p8`	> Write to ***r_wave*** the wave stored in Wave Memory address 3 AND Stores in ***r5*** = ***r6***-1 AND Write ***r_wave*** (the new updated value from memory) to Wave Port 8 at time specified in ***r_time***
`JUMP [STAT_ADDR] -if(S) -wr(r1 op) -op(r1-#1) -uf`	>Jump to START_ADDR if Sign flag (Negative) AND Write ***r1*** = ***r1***-1 (Decrement ***r1***)AND Update ***S*** and ***Z*** Flags

5 ASSEMBLER

The qick_processor python assembler translates assembly language with the format specified in this document to a bit file ready to be loaded to the Program File. This translation process is called assembly.

5.1 STATEMENTS

This section outlines the types of statements that apply to assembly language.
Each statement must be one of the following types:

- An empty statement is one that contains nothing other than spaces, tabs, or formfeed characters. Empty statements have no meaning to the assembler. They can be inserted freely to improve the appearance of a source file or of a listing generated from it.
- A label consists of a symbol ending with a colon (:). When the assembler encounters a label, it assigns the value of the location counter to the label.
- A comment can be inserted to improve the code. Appending a comment at the beginning or the end of the statement by preceding the comment with a double slash (//).
- A machine command statement is a mnemonic representation of an executable instruction to which it is translated by the assembler. It consists of an instruction, optionally followed by operands.
- A directive statement is an instruction to the assembler that not necessarily generates code.

5.2 DIRECTIVES

5.2.1.1 .ALIAS

Alias names for the registers can be created using the .ALIAS directive

ASSEMBLER EXAMPLES:

`.ALIAS addr_aux r0` `REG_WR addr_aux #100`	`> Set the name for Register r0 to addr_aux` `> Stores in addr_aux` **`(r0)`** `value 100`

`.ALIAS step_time r1`
`REG_WR step_time  #256`
`REG_WR r_time op -op(r_time +step_time)`
	`> Set the name for Register r1 to step_time`
	`> Stores in step_time`**`(r1)`** `imm value 256`
	`> Increments r_time in step_time(256)`

5.2.1.2 .CONST

Define Constant Values for the Assembler

ASSEMBLER EXAMPLES:

`.CONST width #256` `REG_WR r0 imm width`	`> Defines CONSTANT width = 256` `> Stores in` **`r0`** `constant width(100)`

`.ALIAS step_time r1`
`.CONST step_time  #256`
`REG_WR r_time op -op(r_time +step_time)`
	`> Set the name for Register r1 to step_time`
	`> Defines CONSTANT step_time=256`
	`> Increments r_time`**`(r1)`** `in step_time(256)`

`.ALIAS r_cnt r0`
`.CONST total_repeat #100`
`REG_WR repeat_cnt imm total_repeat`
`REG_WR repeat_cnt op -op(repeat_cnt-#1) -uf`
`LOOP:`
	`> Set the name for Register r0 to r_cnt`
	`> Create the constant tot_repeat = 100`
	`> Stores in r_cnt`**`(r0)`** `constant tot_repeat(100)`
	`> Decrement r_cnt`**`(r0)`** `and update Flag`

`    REG_WR r1 op -op(r1+#1)`
`JUMP LOOP -wr(r_cnt op) -op(r_cnt-#1) -if(NZ) -uf`
	`> Creates label LOOP`
	`> Increment r1`
	`> Jump if flag not_zero and decrement r_cnt`

NOTE: Alias and CONST replace text, so maybe some errors when are a match in part of the recognition
Label > F_C REATE_WAVES and CONST WAVES gives error…

5.2.1.3 .ADDR

Set the address for the next instruction

ASSEMBLER EXAMPLES:

`.ADDR 100` `REG_WR r1 imm #1`	`> Set the next instruction in address 100` `> Instruction REG_WR in address 100`

`.ADDR 16`
`LABEL_16:`
`REG_WR r0 imm #0`
	`> Set the next instruction in address 16`
	`> Creates the Label in Address 16`
	`> Instruction REG_WR in Address 16`

5.2.1.4 .END

Pro gram can be finished with the .END directive, this will avoid the program counter to go further. This directive add a `JUMP HERE` instruction to the program.

5.3 Machine Instruction Syntax

This section describes the instructions that the assembler accepts. The detailed specification of how the instructions operates is not included.

The following list describes the two main aspects of the qick_processor assembler:

- The 3 banks of registers use a prefix to distinguish them from symbol names. (r, s, or w).
  - General purpose registers start with r
  - Special Function registers start with s
  - Wave Parameters registers start with w
  - Special Function and wave Register have alias names.
- Instructions with two operands use the left one as the destination and the right one as the source.
  - Instructions Options can be written in any order

5.3.1 Operands

Two kinds of operands are generally available to the instructions: register and immediate. The assembler always assumes it is generating code for a 32-bit signed integer values. For some dual operations, the immediate value can be a 24-bit or a 16-bit integer.

5.3.2 Addressing modes

- The Processor has 4 addressing modes for the Data Memory address
  - Literal > The address is an Immediate Value
  - Register > The address is stored in a Register
  - Indexed Literal > The address is the Sum of a Register Value and a Immediate Value
  - Indexed Register > The address is the Sum of 2 registers.

- The Processor has 2 addressing modes for the WaveParam Memory address
  - Literal > The address is an Immediate Value
  - Register > The address is stored in a Register
- The Processor has 2 addressing modes for Branch instructions
  - Literal > The address is an Immediate Value
  - Register > The address is the register **s15** r_addr

5.3.3 Instruction description

This section describes the qick_processor instruction syntax. The assembler generate code for a 32-bit integer values for data and 16-bit values for address.

- All Instructions are UPPERCASE.
- Comments starts with //. In Line comments are allowed.
  - Avoid using ":" in the comment, it will be interpreted as a LABEL
- Labels end with ":" Contains numbers, UPPERCASE, lower case and "_" symbol. No space is allowed in Labels.
- Literal Values start with "#" symbol.
  - Signed Values start with "#".
  - Unsigned Values start with "#u".
  - Binary Values start with "#b".
  - Hexadecimal Values start with "#h".
- General Purpose registers start with r
- Special Function Registers start with s
- Wave Parameter registers start with w
- Conditions are UPPERCASE
- Address should go between square brackets "[]"
- Literal Address start with the & symbol.
- Literal Port Values are just an integer.
-
- Address can only be General Purpose registers
- Ports start with p
- Time starts with @
- Instruction options, second instructions and register alias are lowercase.

CORRECT	INCORRECT
REG_WR r0 imm #0	reg_wr r0 imm #0
// Comment	/* Comment (Only one /)
REG_WR r0 imm #0 **//** Set r0 to Zero	REG_WR r0 imm #0 --Set r0 to Zero (Not //)
LOOP_START:	:LOOP_START (Not ends with :)
OPTION_2_reset:	:OPTION_#2 (# not allowed)
REG_WR r0 op -op(w0 AND #b101)	REG_WR r0 op -op(wo AND #b5) (with #b only 0 and 1)
REG_WR **r0** imm #0	REG_WR reg0 imm #0 (is r0)
REG_WR r0 op -op(w0+**s1**)	REG_WR r0 op -op(W0+S1) (w and s lowercase)
REG_WR w_freq op -op(**w0**+#1)	REG_WR r_freq op -op(W0+#1) (Alias is r_freq, reg is w0)
JUMP LABEL -if(**NZ**)	JUMP [LABEL] -if(nz) (no use of []is NZ)
DMEM_WR **[r1]** imm #b0110100	DMEM_WR &1 imm #b0110100 (Not [])
WMEM_WR [**&0**]	WMEM_WR [#0] (Not &)
DMEM_WR [**r2+&2**] op -op(w_freq)	DMEM_WR [s2+&2] op -op(w_freq) (Address only dreg)

`DPORT_RD` **`p0`**	`DPORT_RD port0 (is p0)`
`DPORT_WR` **`p1`** `imm 5`	`DPORT_WR &1 imm #5 (is p1, dataport not use #)`
`DPORT_WR` **`p1`** `reg r3`	`DPORT_WR` **`p1`** `op -op(r3) (only lit or reg as source)`
`WPORT_WR wmem [r2]` **`p2`**	`WPORT_WR wmem [s2] p2 (Address only dreg)`
`WPORT_WR r_wave p3` **`@99`**	`WPORT_WR r_wave @99 (Missing Port)`
`WPORT_WR wmem [&2] p2` **`@125`**	`WPORT_WR wmem p2 @125 (Missing wmem address)`
`REG_WR` **`r_wave wmem`** `[&3]` **`-wr`**`(r5 op)` **`-op`**`(r6-#1)`	`REG_WR w_wave wmem [&3] -WR(r5 OP) -OP(r6-#1)`

5.4 Instruction Description

Instructions are 72-bit wide. The 72 Bits are divided in 16-bits OP_CODE and 56-bits OP_DATA.

TYPE	OP_CODE												OP_DATA			RD
	15 14 13	12	11 10	9	8	7	6	5	4	3	2	1 0	55 50 45	44 39	38 31 30 23 22 15 14 7	6 0
ALU	Header	AI	DF	COND			reg_src		uf	alu_op			Addr[0]	Addr[1]	Data Source	Reg Dest
DATA	Header	AI	DF	COND			T	DI	uf	wr	rdi	alu_op	Addr/PData	Port	Time / RData Source	Reg Dest
WAVE	Header	AI	DF	Ww	Ps	Wp	T	TI	uf	wr	rdi	alu_op	AddrWmem	Port	Time / Data Source	Reg Dest
CFG	Header	AI	DF	CFG					uf	wr	rdi	alu_op	Data Source			Reg Dest
CTRL	Header	AI	DF	Operation					Control				Data Source			

- AI: Address/PData Immediate
- DF: Data Format
- COND: Condition (Combination of Flags)
- Ww: Write Wave Memory
- Ps: Port Source W(0-Mem 1-Reg)
- Wp: Write Port
- reg_src: Source for Register Write
- T: Port Type (0-Wave 1-Data)
- DI: Data Source (0-ALU 1-Imm)
- TI: Time Source (0-r_time 1-Imm)
- uf: Update Flag
- wr: Write Register
- rdi: Register Data Imm or ALU
- alu_op: Arithmetic Operation

DF		Imm Size
0	0	RFU
0	1	16 Bits
1	0	24 Bits
1	1	32 Bits

reg_src		Source
0	0	ALU
0	1	Data Mem
1	0	RFU
1	1	Imm

COND			Condition
0	0	0	ALWAYS
0	0	1	Zero
0	1	0	Sign (Negative)
0	1	1	Not Zero
1	0	0	Not Sign (Not Negative)
1	0	1	FLAG
1	1	0	NO FLAG
1	1	1	RFU

Figure 11: Instruction OPCODES

Header : These 3 bits indicate the type of instruction. There are 8 Types of instructions as shown in Table 16: Header Instruction Type.

Header			Type	Description
0	0	0	CFG	Configuration Instructions
0	0	1	BRANCH	Program Branching Instructions
0	1	0	RFU	Reserved for Future Use
0	1	1	RFU	Reserved for Future Use
1	0	0	REG_WR	Register Write Instructions
1	0	1	MEM_WR	Memory Write Instructions
1	1	0	PORT_WR	Port Write Instructions
1	1	1	CTRL	tProc/ Co-Processor Control Instructions

Table 16: Header Instruction Type

AI [Address Immediate]: This bit indicates if the Address Source [0] (Bits OPData [55:45]) is an immediate value or a register stored value. 0-Register 1-Immediate

DF [Data Format]: These two bits are used to define the Data Source format. A Zero indicates two registers will be used. A One indicates two registers and an Immediate 16Bits value will be used. A Two indicates one register and an Immedite 24Bits will be used. A Three indicates an Immediate 32Bits value is used. As shown in Table 18: Data Source Format

DF		Data_Source							
		38	31	30	23	22	15	14	7
1	1	Immediate DT							
1	0	rsD[0]		Immediate DT					
0	1	rsD[0]		rsD[1]		Immediate DT			
0	0	rsD[0]		rsD[1]		.			

Table 17: Data Source Format

COND[Condition]: Some Data instruction and Branch instructions contains a condition field wich determines whetheIndicate the Condition for Instruction execution. Some Instructions are executed only of the Condition is md

reg_src [Register Data Source]: Indicate the source of the data to the register to be written.

uf [Update Flag] :

wr [Register Data Immediate] : wr Indicate if theres is a optional write register operation.

rdi [Write Register] : RDI indicates if the data source to the register is immediate or Alu 0-Register 1-Immediate.

alu_op [Arithmetic Logic Unit Operation] : Indicates the operation to be done by the ALU. If is a write_reg operation there are 16 operations, otherwise (for a second option write reg) 4 operations are available.

5.4.1 Configuration Instructions.

Instruction with HEADER = 000. This set of instructions does not change the value of any data (Register, Memory, or Port). Just the Flags.

5.4.1.1 NOP

INTRUCTION	HEADER			AI	DF		COND			SO	TO	uf	Opt																		
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	55	50	45	44	39	38	31	30	23	22	15	14	7	6	0
NOP	0	0	0	0	0		0			0		0	0				0					0								0	

This instruction executes NO OPERATION. It takes One clock cycle

ASSEMBLER EXAMPLE:

```
NOP                                    > No Operation
```

5.4.1.2 TEST

INTRUCTION	HEADER			AI	DF		COND			SO	TO	uf	wr	rdi	alu_op																
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	55	50	45	44	39	38	31	30	23	22	15	14	7	6	0
TEST	0	0	0	0	1	0	0			0		1	0	0	alu_op		0					rsD[0]		Immediate DT						0	
TEST	0	0	0	0	0	1	0			0		1	0	0	alu_op		0					rsD[0]		rsD[1]		0				0	

This instruction performs an ALU operation and update the FLAG values, but to result is written. The operation can be done between 2 registers or a register and an Immediate Value. Possible ALU operations are +, -, AND and ASR.

ASSEMBLER EXAMPLES:

```
TEST -op(r3 - #3)                > Update the flags with the Instruction r3-3
```

TEST -op (r4 AND #b11)	> Update the flags with the Instruction r3 AND 3

5.4.2 Register Instructions

Instruction with HEADER = 100. This set of instructions change Register values (General Purpose, Special Function and WaveParam)

5.4.2.1 REG_WR

INTRUCTION	HEADER			AI	DF		COND			Source		UF	SDI				Address Source					Data Source								Reg Dest	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	55	50	45	44	39	38	31	30	23	22	15	14	7	6	0
REG_WR	1	0	0	0	DF		Cond			0	0	alu_op					0					Data Source								Reg Dest	
REG_WR	1	0	0	AI	DF		Cond			0	1	uf	0	0	alu_op		rsD[0]			rsA[1]		Data Source								Reg Dest	
REG_WR	1	0	0	AI	DF		Ww	Ps	Wp	1	0	uf	wr	rdi	alu_op		AddrWmem			Port		Data Source								Reg Dest	
REG_WR	1	0	0	0	DF		Cond			1	1	uf	0	0	alu_op		0					Data Source								Reg Dest	

This instruction performs a register write, the destination register can be a ***sreg***, a ***dreg***, ***wreg*** or ***r_wave.*** Data Source can be an ALU operation (00), a 32-bit Data Memory (01), a 128-bit WaveParam (10) or an Immediate Value(11).

If Source is ALU operation (00), no SDT (Second Data Task) can be done. The Operation can be done between 2 registers (***dreg***, ***sreg*** or **wreg**) or a register(***dreg***, ***sreg*** or **wreg**) and a 24-bit Immediate Value. This instruction is conditional

If Source is Data Memory (01) or Immediate Value(11), a TEST operation can be done as a SDT. This instruction is conditional

If Source is WaveParam Memory(10), the destination register is the ***r_wave***. A Second Data Instruction can be done. Write Wave, and Write Port can be done as a Second Wave (SWT) and Port (SPT) Task. The SPT Source can be the ***r_wave*** or the WaveParam Memory Data. This instruction is NOT conditional

Sources

- op > ALU Source
- imm > Immediate Value Source
- dmem > Data Memory Source (Read Data memory Command)
- wmem > WaveParam Memory Source (Read WaveParam memory Command)
- label > Used to store in a register (usually **r_addr**) the address of a label in the program memory.

ASSEMBLER EXAMPLES:

REG_WR r0 imm #0	> Stores in **r0** value 0
REG_WR r1 op -op(rand)	> Stores in **r1** random number
REG_WR r2 op -op(r1 ASR #1)	> Stores in **r2** Arithmetic shift right once r1
REG_WR r3 dmem [&10]	> Stores in **r3** Data Memory Address 10
REG_WR r4 dmem [r2 + &10]	> Stores in **r5** Data Memory Address r2+10
REG_WR r_wave wmem [&0]	> Stores in **r_wave** WaveParam Memory Address 0
REG_WR r_addr label PROC_1	> Stores in **r_addr** address of label PROC_1.

If a register wave is readed from memry a second task instruction can be done to update a new value in a register.
REG_WR r_wave wmem [&3] -wr(w1 imm) #123
This instruction will load the waveform on address3, but will copy the NEW 123 phase instead of the one in the memory.
and modifies the phase

5.4.3 Memory Instructions

Instruction with HEADER = 101. This set of instructions write Memory values (Data and WaveParam)

Do not use wreg as an address.

5.4.3.1 DMEM_WR

INTRUCTION	HEADER			AI	DF		COND			SO	TO	UF	SDI				Address Source					Data Source								Reg Dest	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	55	50	45	44	39	38	31	30	23	22	15	14	7	6	0
DMEM_WR	1	0	1	AI	DF		COND			0	DI	uf	wr	rdi	alu_op		rsA[0]			rsA[1]		Data Source								Reg Dest	

This instruction performs a Data Memory Write instruction, the destination address is rsA[0]+rsA[1], where rsA[0] can be a register address or a literal value, depending on the AI bit. Register rsA[1] can only be a ***dreg*** or a s***reg*.**

A Write Data Register operation can be done as a (Second Data Task) SDT.

This instruction is conditional.

The immediate address is stored in rsA[0], so the maximum immediate address is 11 bits.

ASSEMBLER EXAMPLES:

`DMEM_WR [&0] imm #10`	> Stores in Data Memory Address 0, Value 10
`DMEM_WR [r1] op -op(r0 + #1)`	> Stores in Data Memory Address pointed by **r1** value of the operation **r0** + 1
`DMEM_WR [r1+r3] op -op(r_freq)`	> Stores in Data Memory Address pointed by **r1+r3**, value of the freq parameter(w0)
`DMEM_WR [r1+&4] imm #10 -wr(r5 op) -op(r2+r3)`	> Stores in Data Memory Address pointed by **r1+4**, value 10 >>SDT: Writes in **r5**=**r2**+**r3**

5.4.3.2 WMEM_WR

INTRUCTION	HEADER			AI	DF		COND			SO	TO	UF	SDI				Address Source					Data Source								Reg Dest	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	55	50	45	44	39	38	31	30	23	22	15	14	7	6	0
WMEM_WR	1	0	1	AI	DF		1	Ps	Wp	1	TI	uf	wr	rdi	alu_op		AddrWmem			Port		Time / Data Source								Reg Dest	

This instruction performs a WaveParam Memory Write instruction, the destination address can be a Literal value or a register value, depending on the AI bit. The Source of the data is always the 128-bit **r_wave** register.

A Write Data Register or TEST operation can be done as a (Second Data Instruction) SDT.

A Write Port can be done as a Second Port Task (SPT). The SPT Source can be the ***r_wave*** or the previous WaveParam Memory Data on the address selected (Before the Write).

This instruction is NOT conditional

ASSEMBLER EXAMPLES:

`WMEM_WR [&0]`	> Stores in WaveParam Memory Address 0, the **r_wave** value
`WMEM_WR [r1] -wr(r_freq op) -op(r_freq+#10)`	> Stores in WaveParam Memory Address pointed by **r1** the **r_wave** value >>SDT: Increment r_freq value in 10. The value stored in memory is previous the increment.
`WMEM_WR [&3] -wr(r_gain op) -op(r_gain+#25) -wp(r_wave) -p3`	> Stores in WaveParam Memory Address 3 the **r_wave** value >>SD: Increment the gain value in 25 >>SPT: Writes the **r_wave** to Waveport 3

t

5.4.4 Port Instructions

Instruction with HEADER = 110. This set of instructions Operates with Ports (Data and WaveParam)

5.4.4.1 DPORT_WR

INTRUCTION	HEADER			AI	DF		COND			SO	TO	UF	SDI				Address Source					Data Source								Reg Dest	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	55	50	45	44	39	38	31	30	23	22	15	14	7	6	0
DPORT_WR	1	1	0	0	DF		0	0	1	0	DI	uf	wr	rdi	alu_op		0			Port		Data Source								Reg Dest	

CHANGE DATA BITS, 55:45 are IMMEDIATE VALUES

This instruction performs a Data Port Write. The destination port is always a literal value, the time can be a immediate value or **r_time** (**s14**). The Source Data can be a Literal or a reg

A Write Data Register or TEST operation can be done as a (Second Data Instruction) SDT. If a SDT operation is done, TIME should be the r_time.

The immediate value should be less than 11 bits. (bits 55:45)

The immediate value for port write in this case should go without #. The # is used for the imm value of the SDT

This instruction is NOT conditional

ASSEMBLER EXAMPLES:

`DPORT_WR p0 imm 1 @125`	> Writes the value 1 in port 0 at time 125
`DPORT_WR p1 imm 3 -wr(r1 imm) #2`	> Writes the value 3 in port 1 at the time specified in **r_time**. >>SDT: Write imm value 2 in r1
`DPORT_WR p2 imm 5 -wr(r1 op) -op(r1-#1) -uf`	> Writes the value 5 in port 2 at the time specified in **r_time**. >>SDT: Decrements r1 and update the flags.
`DPORT_WR p0 reg r3 @100`	> Writes the value on register r3 in port 0 at time 100
`DPORT_WR p1 reg r4 -wr(r1 imm) #7`	> Writes the value on register r4 in port 1 at the time specified in **r_time**. >>SDT: Write imm value 7 in r1
`DPORT_WR p2 reg r5 -wr(r1 op) -op(r1+r2)`	> Writes the value of register r1 at the output port 1 >>SDT: Write Register r1=r1+r2.

5.4.4.2 TRIG

Triggers a port. Set the digital value f a trigger port to 1 or 0

If a SDT operation is done, TIME should be the r_time.

ASSEMBLER EXAMPLES:

`TRIG p0 set @150`	> Writes a digital ONE in the trigger port 0 at time 150
`TRIG p1 clr -wr(r1 imm) #2`	> Writes a digital ZERO in the trigger port 1 at the time specified in **r_time**. >>SDT: Write imm value 2 in r1
`TRIG p2 set`	> Writes a digital ONE in the trigger port 2 at the time specified in **r_time**.

5.4.4.3 DPORT_RD

INTRUCTION	HEADER			AI	DF		COND			SO	TO	UF	SDI				Address Source					Data Source								Reg Dest	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	55	50	45	44	39	38	31	30	23	22	15	14	7	6	0
DPORT_RD	1	1	0	.	DF		.	.	0	0	0	uf	wr	rdi	alu_op		.			Port		Data Source								Reg Dest	

This instruction performs a Data Port Read. The destination registers are always Special registers **port_l**(**s8**) and **port_h** (**s9**). The source port is a literal value.

A Write Data Register or TEST operation can be done as a (Second Data Instruction) SDT.

This instruction is NOT conditional

ASSEMBLER EXAMPLES:

`DPORT_RD p0`	`> Reads the value of the Port 0.`
`DPORT_RD p1 -wr(r15 op) -op(r1-#1) -uf`	`> Reads the Port 1` `>> SDT: Write in register` **`r15=r1-1`** `and updates flags.`

5.4.4.4 WPORT_WR

INTRUCTION	HEADER			AI	DF		COND			SO	TO	UF	SDI				Address Source					Data Source								Reg Dest	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	55	50	45	44	39	38	31	30	23	22	15	14	7	6	0
WPORT_WR	1	1	0	AI	DF		Ww	Ps	1	1	TI	uf	wr	rdi	alu_op		AddrWmem			Port		Time / Data Source								Reg Dest	

This instruction performs a WaveForm Port Write. The destination port is always a literal value. The time can be the special register **r_time** (**s14**), or an Immediate Value, depending on the TI bit. The Source Data can be the 128-bit **r_wave** register or a WaveParam Memory value, addressed literal or registered.

If **r_time** is used as a time source, a Write Data Register or TEST operation can be done as a (Second Data Instruction) SDT.

This instruction is NOT conditional

ASSEMBLER EXAMPLES:

`WPORT_WR p0 r_wave`	`> Writes the` **`r_wave`** `in port 0 at the time specified in` **`r_time`**`.`
`WPORT_WR p1 r_wave @125`	`> Writes the` **`r_wave`** `in port 0 at time 125`
`WPORT_WR wmem [&2] p2 -wr(r1 op) -op(r1+#1)`	`> Writes the value of WaveParam Memory address 2 in port 2 at the time specified in` **`r_time`** `>> SDT: Increments r1. No flag update.`

5.4.5 Branch Instructions

Instruction with HEADER = 001. This set of instructions operates the Program Counter (PC).

5.4.5.1 JUMP

INTRUCTION	HEADER			AI	DF		COND			SO	TO	UF	SDI				Address Source					Data Source								Reg Dest	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	55	50	45	44	39	38	31	30	23	22	15	14	7	6	0
JUMP	0	0	1	AI	DF		COND			0	.	uf	wr	rdi	alu_op		AddrJMP			.		Data Source								Reg Dest	

This instruction performs a change in the PC, making the program jump to a specified program address. The destination address can be a literal value or the special register **r_addr** (**s1r**), depending on the AI bit.

A Write Data Register or TEST operation can be done as a (Second Data Task) SDT.

This instruction is conditional.

If address is a label the [] should not be written.

The assembler for this instruction has 4 reserved words.

- ***"HERE"*** is the same address than the JUMP instruction. Is used to stay on the same address (To create a Wait Intruction)
- ***"PREV"*** is the previous instruction. Jump to the address PC-1
- ***"NEXT"*** goes to the next instruction. Jump to the address PC+1
- ***"SKIP"*** skip the next instruction. Jump to the address PC+2
- "**r_addr**" is used to jump to the address specified by the special function register **r_addr**(**s15**)

ASSEMBLER EXAMPLES:

`JUMP [%0]`	`> Unconditional jump to Address 0.`
`JUMP HERE -if(NEC)`	`> Stay in the same address until a external condition flag is set.`
`JUMP LOOP -if(NZ) -wr(r1 op) -op(r1-#1) -uf`	`> Jump to LOOP label if flag is NZ.` `>> SDT: Decrement r1 and updates the Flags.`

5.4.5.2 CALL

INTRUCTION	HEADER			AI	DF		COND			SO	TO	UF	SDI				Address Source					Data Source								Reg Dest	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	55	50	45	44	39	38	31	30	23	22	15	14	7	6	0
CALL	0	0	1	AI	DF		COND			1	0	uf	wr	rdi	alu_op		AddrJMP			.		Data Source								Reg Dest	

This instruction performs procedure CALL. Stores the PC in the Stack and jumps to the specified address. The destination address can be a literal value or the special register **r_addr** (**s1r**), depending on the AI bit.

A Write Data Register or TEST operation can be done as a SDT.

This instruction is conditional.

5.4.5.3 RET

INTRUCTION	HEADER			AI	DF		COND			SO	TO	UF	SDI				Address Source					Data Source								Reg Dest	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	55	50	45	44	39	38	31	30	23	22	15	14	7	6	0
RET	0	0	1	AI	DF		.			1	1	uf	wr	rdi	alu_op		.			.		Data Source								Reg Dest	

This instruction RETURNS from a procedure CALL.

A Write Data Register or TEST operation can be done as a SDT.

This instruction is NOT conditional

5.4.6 Peripherals Control Instruction

Instruction with HEADER = 111. This set of instructions allow the user to control the Peripherals Blocks.

5.4.6.1 TIME

INTRUCTION	HEADER			AI	DF		Operation					Control					Address Source					Data Source								Reg Dest	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	55	50	45	44	39	38	31	30	23	22	15	14	7	6	0
TIME	1	1	1	AI	DF		e	d	c	b	a	0	0	0	0	1	.			.		Data Source									

TODO: CHANGE HEADER VALUE TO 010 add syntax

This instruction perform time related instructions. It can modify the value of `ref_time and abs_time`. It has 4 diferent options.

- Reset the absolute time `abs_time` = 0

- Set Reference time to a specific value (`ref_time` = Source Data)
- Increase Reference Time a certain value (`ref_time` = `ref_time` + Source Data)
- Set the Time comparator for Time Flag.

ASSEMBLER EXAMPLES:

`TIME rst`	`> Reset absolute time.`
`TIME set_ref r2`	`> Set the reference time to value in r2`
`TIME inc_ref #15750`	`> Increment the reference time 15750(ref_time = ref_time + 15750)`
`TIME set_cmp #16800`	`> Set the Time comparator to set the flag after the time 16800`

5.4.6.2 FLAG

INTRUCTION	HEADER			AI	DF		Operation					Control					Address Source					Data Source								Reg Dest	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	55	50	45	44	39	38	31	30	23	22	15	14	7	6	0
COND	1	1	1	AI	DF		e	d	c	b	a	0	0	0	1	0	.			.		Data Source									

TODO: CHANGE HEADER VALUE TO 010 add syntax

This instruction clears or set the external condition flag.

The flag is set with this command, with an arithmetc operation ends and when NEW data is written in some PORT. If the flag was set by a PORT. The only way to clear is by reading the data from the port with the Dr_waveinstruction.

ASSEMBLER EXAMPLES:

`FLAG set`	`> set the External condition flag.`
`FLAG clear`	`> Clears the External condition flag.`

5.4.6.3 ARITH

INTRUCTION	HEADER			AI	DF		Operation					Control					Data Source													Reg Dest	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	55	50	45	44	39	38	31	30	23	22	15	14	7	6	0
ARITH	1	1	1	AI	DF		e	d	c	b	a	0	0	1	0	0		C		D		A		B							

TODO: CHANGE HEADER VALUE TO 010 add syntax

This instruction perform an arithmetic operation with the FPGA DSP. Make (D ± A) * B ± C Arithmetic operation in 2 clock cycles. 64-bit result value is stored in Special Function Registers s2 and s3.

With values a, b, c, d, e of the Operation Field in the OP_Code, user selects the operation. A List of operation codes are shown in Table 18: ARITH Operation Codes. There are 16 Operations

The operation is encoded with Leters P for PLUS, M for Minus and T for times. (D+A)*B-C is PTM D A B C

OPERATION >

Value				Operation	
0	0	0	0	A*B	T
0	0	0	1	A*B + C	TP
0	0	1	0	A*B - C	TM
0	0	1	1	(D+A)*B	PT
0	1	0	0	(D+A)*B+C	PTP
0	1	0	1	(D+A)*B-C	PTM
0	1	1	0	(D-A)*B	MT
0	1	1	1	(D-A)*B+C	MTP
1	0	0	0	(D-A)*B-C	MTM

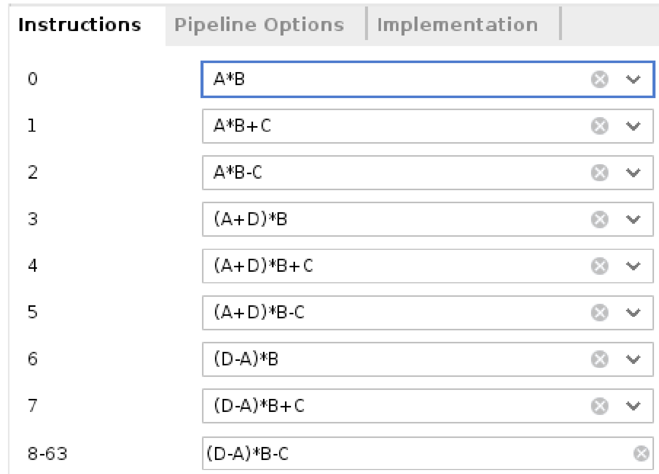


Table 18: ARITH Operation Codes

OPERATION		Assembler Command	Operation Executed
T	A * B	`ARITH T r1 r2`	r1 * r2
TP	A * B + C	`ARITH TP r1 r_freq rand`	r1 *r_freq+rand
TM	A * B - C	`ARITH TM r1 s2 r3`	r1 * s2 – r3
PT	(D+A) * B	`ARITH PT r2 r1 r_gain`	(r2 + r1) * r_gain
MT	(D-A) * B	`ARITH PT r1 r2 r3`	(r1 – r2) * r3
PTP	(D+A) * B + C	`ARITH PTP r1 rand r3 r4`	(r1 + rand) * r3 + r4
PTM	(D+A) * B - C	`ARITH PTM r1 r2 r3 r4`	(r1 + r2) * r3 - r4
MTP	(D-A) * B + C	`ARITH MTP r1 s2 w3 r4`	(r1 - s2) * w3 + r4
MTM	(D-A)*B-C	`ARITH MTM r1 r2 r3 r4`	(r1 - r2) * r3 - r4

ASSEMBLER EXAMPLES:

`ARITH T r1 r2`	`> Makes r1 Times r2.`
`ARITH PT r1 s2 w3`	`> Makes (r1 Plus s2) Times w2 (r1+s2)*w2`
`ARITH MTP r1 s2 w3 r4`	`> Makes (r1 Minus s2) Times w3 Plus r4 (r1-s2)*w3+r4`

`ARITH TM r1 r2 r3`
`REG_WR s_conf imm flag_div`
`JUMP HERE -if(NF)`
`REG_WR s_conf imm src_arith`
	`> Makes r1 Times r2 Minus r3 => r1*r2-r3`
	`> Select Arith End as source for FLAG condition.`
	`> Wait until Arithmetic Operation ends.`
	`> Select Arith Data as source for CORE_R_DT`

5.4.6.4 DIV

INTRUCTION	HEADER			AI	DF		Operation					Control					Data Source													Reg Dest	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	55	50	45	44	39	38	31	30	23	22	15	14	7	6	0
DIV	1	1	1	AI	DF		0	0	0	0	1	0	1	0	0	0						NUM		DEN							

TODO: CHANGE HEADER VALUE TO 010 add syntax

This instruction configures the Division Unit Co-Processor to make a division. The destination registers are always Special registers **div_q**(**s4**) for the quotient and **div_r** (**s5**) for the remainder of the division.

The source data for the Numerator is always a register and the denominator can be a register or a Literal Value.

ASSEMBLER EXAMPLES:

`DIV r1 r2`	`> Makes r1 / r2.`
`DIV r1 #100`	`> Makes r1 / 100`
`DIV #100 r1`	`> Makes r1 / 100 (Immediate is always the denominator)`

`DIV r1 r2`
`REG_WR s_conf imm flag_div`
`JUMP HERE -if(NF)`
	`> Makes r1 / r2`
	`> Select Division End as source for FLAG condition.`
	`> Wait until Division ends.`

5.4.6.5 NET

INTRUCTION	HEADER			AI	DF		Operation					Control					Data Source													Reg Dest	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	55	50	45	44	39	38	31	30	23	22	15	14	7	6	0
NET	1	1	1	AI	DF		0	0	0	0	1	1	0	0	1	0		C		D		A		B							

TODO: CHANGE HEADER VALUE TO 011 add syntax

ASSEMBLER EXAMPLES:

`NET get_net`	`> Count nodes in network.`
`NET set_net`	`> Configure all nodes in Network.`
`NET sync_net`	`> Syncronice all nodes in Network`
`NET get_st`	`> Read data from node in network.`

5.4.6.6 COM

5.4.6.7 PA / PB

INTRUCTION	HEADER			AI	DF		COND			ADDR		OPERATION					Data Source													Reg Dest	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	55	50	45	44	39	38	31	30	23	22	15	14	7	6	0
PA	0	1	1	0	0	1		COND		1	0			Operation			.	C		D		A		B			.			.	
PB	0	1	1	0	0	1		COND		1	1			Operation			.	C		D		A		B			.			.	

This instruction generates the outputs to use an external peripheral. The operation is an integer value between 0 and 32. All the values should be registers.

- Conditional instruction.
- NOT a Timed instruction.

Restricion> C and D can not be wreg

Syntax:

`PA/PB OP A {B} {C} {D} {cond}`

ASSEMBLER EXAMPLES:

`PA 1 r1`	`> Peripheral A Operation 1, One Input`
`PB 2 r1 rand -if(Z)`	`> Peripheral B Operation 2, Two Inputs, if Flag Zero`
`PA 3 r1 rand w_freq`	`> Peripheral A Operation 3, Three Inputs`
`PB 7 r1 rand w_freq r2`	`> Peripheral B Operation 7, Four Inputs`

5.4.7 Multi Instruction Commands

5.4.7.1 WAIT

This command inserts two instructions in order to wait until the desired time specified. WAIT @50 the instruction automatically subtracts the time needed for clock change domain. It will have an error of 2 clocks.

ASSEMBLER EXAMPLES:

```
WAIT @160
     TEST -op(tuser - #150)              > WAIT TO 150, using the internal ALU
     JUMP PREV -if(S)
```

6 PYTHON INTERFACE

When the qick lbrary is imported, also the qick_processor driver is imported.

```
The qick_processor driver > /qick_lib/qick/drivers/tproc.py

##### Load FGPA BitStream
soc = QickSoc('./zcu_216_net.bit')
```

Commands in

time_reset()	Reset time
time_update()	Update time value
start()	Start the qick_processor
stop()	Stop the qick_processor
reset()	Reset the qick_processor
core_start()	Start the Core
core_stop()	Stop the Core
single_read(mem_sel, addr)	Read single Data (32-bit) from memory
single_write(mem_sel, addr)	Write single Data (32-bit) to memory
load_mem(mem_sel, buff_in)	Write to memory using DMA
read_mem(mem_sel, addr, lenght)	Read from memory using DMA
load_Pmem(mem_dt)	Load Program memory

ASSEMBLER EXAMPLES:

```
soc = QickSoc('./zcu_216_net.bit')   > Load FPGA
soc.tproc.Load_PMEM(p_bin)           > Load Program Memory
soc.tproc.start                      > Start the qick_processor
soc.tproc.stop                       > Stop the qick_processor
```

When the qick lbrary is imported, also the assempler is imported.

```
The assembler script > /qick_lib/qick/tprocv2_assembler.py
```

The class Assembler

- file_asm2bin(file) > generates the executable binary from an assembler file
- str_asm2bin(str) > generates the executable binary from an assembler string variable
- file_asm2list(file) > generates instruction list from an assembler file
- str_asm2list(str) > generates instruction list from an assembler string variable
- list2asm(list, alias) > generates the asm code from an instruction list.
- list2bin(list, alias) > generates the binary from an instruction list.

The program to be executed in the qick_processor core can be provided from 3 sources.

- Assembler file > Edited externally and and assembled with python, processed and assembled.
- String variable > Edited in the same Jupyter notebook and then processed and assembled
- Command List > A list of commands and label address needed to generate the binary code.

COMMAND LIST EXAMPLES:

```
prog_list = []
Dict_Label = { 'r_addr':'s15'    }
prog_list.append({'CMD':"REG_WR"    , 'DST':'r1', 'SRC':'op'   ,'OP':'MSH s7' } )
prog_list.append({'CMD':'WPORT_WR' , 'DST':'4' , 'SRC':'wmem','ADDR':'r0' }    )
prog_list.append({'CMD':'DPORT_WR' , 'DST':'0' , 'SRC':'reg' ,'DATA':'r0' }    )
Dict_Label['END'] = '&' + str(len(prog_list)+1)
prog_list.append({'CMD':'JUMP'      , 'LABEL':'END' } )

p_bin          = Assembler.list2bin(prog_list, Dict_Label)
p_asm          = Assembler.list2asm(prog_list, Dict_Label)
soc.tproc.Load_PMEM(p_bin)
print(p_asm)
```

6.1 LABELS

When assembled (linked) the Labels are converted to immediate values. In order for the function list2asm to recover the immediate value to a Label in the generated assembler file the Label should start wih 'F_' or 'S_' or 'T_'
Will Generate the asm code and the binary for the memory>

```
INIT:
     REG_WR r1 op -op(MSH s7)
     WPORT_WR p4 wmem [r0]
END:
     JUMP END
```

ASSEMBLER EXAMPLES:

```
soc = QickSoc('./zcu_216_net.bit')             > Load FPGA
asm = """                                      > write ASM Program in asm
// TEST program                                String variable
// Write CORE_W_DT SREG
REG_WR s12 imm #12
DPORT_WR p0 imm 1
.END
"""
p_txt, p_bin = Assembler.str_asm2bin(asm)
                                               > get binary from a string
soc.tproc.Load_PMEM(p_bin)                     > Load Program Memory
```

The instruction list is a list of dictionary, with the instructions to execute.

The Keys for the instructions are:

Dictionary		
Description	Key	Example
Command	CMD	'CMD':"REG_WR"
Destination	DST	'DST':"r_wave"
Data Source	SRC	'SRC':"wmem"
Operation	OP	'OP':"r2-r1"

Literal Value	LIT	'LIT':"5"
Label	LABEL	'LABEL':"INIT"
Memory Address	ADDR	'ADDR':"&1"
Out Port for Write Port Instructions	PORT	'PORT':"1"
Data used for DPORT instruction	DATA	'DATA':"1"
Time value for Write Port Instructions	TIME	'TIME':"100"
Conditional Execution	IF	'IF':"Z"
Update Flag	UF	'UF':"1"
Write Register as SDT	WR	'WR':"r1 imm"
Write Port as SPT	WP	'WP':"r_wave"
Write Wave Memory as SWT	WW	'WW':"-ww"
DIV Numerator	NUM	'NUM':"r1"
DIV Denominator	DEN	'DEN':"r2"
Custom Operation	C_OP	'C_OP':"5"
Custom Operation Register 1	R1	'R1':"r1"
Custom Operation Register 2	R2	'R2':"rand"
Custom Operation Register 3	R3	'R3':"w_freq"
Custom Operation Register 4	R4	'R4':"zero"
Line Number (For debugging)	LINE	'LINE':"2"

Table 19: Instruction List Dictionary Key values

7 ARCHITECTURE DETAILS

Type 0 is Data, Type 1 is Wave

All qick_processor instruction are 1 Cycle, with the option of executing more than one task per instruction. For some instruction combination the processor can stall for one or 2 cycles to complete the previous task.

T_clk should be FATSER than c_clk. For cdc

The LIFO for CALL-RET is depth 8.. So 8 calls can be anidated…

- When branching, 2 clock cycles are needed to empty the pipeline.
- When a conditional instruction is executed, condition should be calculated previously, and it takes 2 clock cycles.

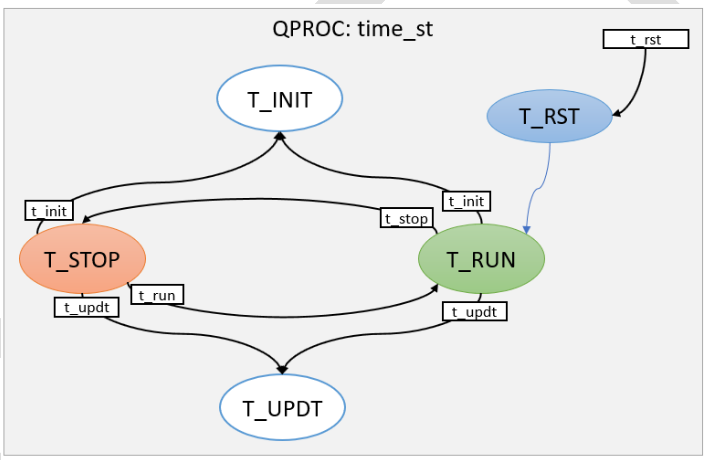


Figure 12: Time Control State Machine

CORE_START > When the *core_start* event is generated the core is reset and goes to the c_run state where the core is running the program stored in the PMEM.

CORE_STOP > When the *core_stop* event is generated the core is stopped. All the registers values remain but the PC does not increment and no instruction is executed.

RUN > This event changes the state of the core from C_STOP to C_RUN.

When the core is reset, the Program Counter (PC) is set to address 0, all the FIFOS are flushed, all the peripheral status bits (new data and ready) are reset , the flags are set to 0, and all the sreg, dreg and wreg are clear. The LFSR is not cleared.

7.1.1 Core Control

CORE_START > This event is triggered by the start, and the core_start Commands. When the *core_start* event is generated the core is reset and goes to the c_run state where the core is running the program stored in the PMEM.

CORE_STOP > When the *core_stop* event is generated the core is stopped. All the registers values remain but the PC does not increment and no instruction is executed.

RUN > This event changes the state of the core from C_STOP to C_RUN.

When the core is reset, the Program Counter (PC) is set to address 0, all the FIFOS are flushed, all the peripheral status bits (new data and ready) are reset , the flags are set to 0, and all the sreg, dreg and wreg are clear. The LFSR is not cleared.

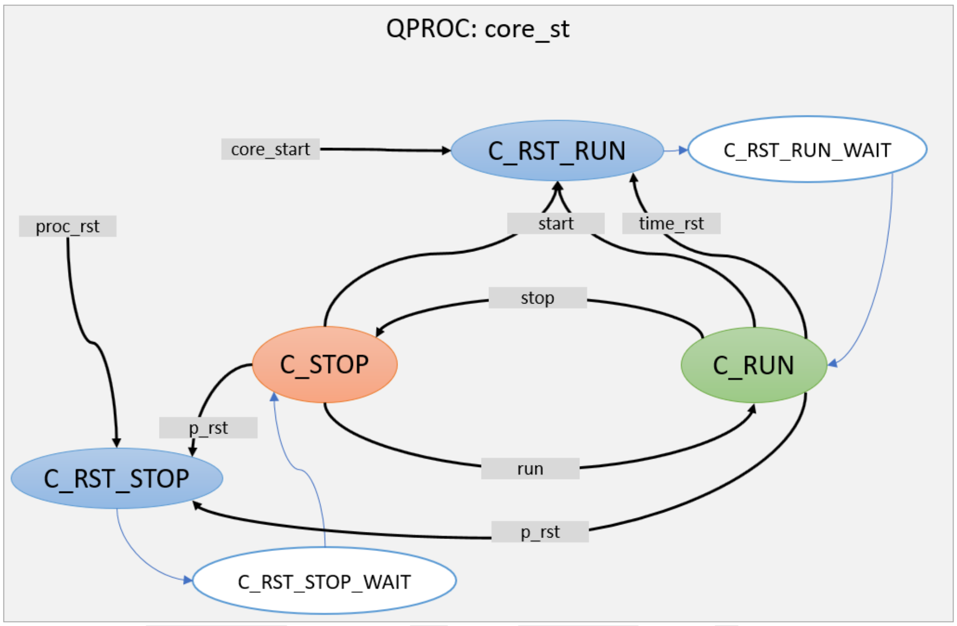


Figure 13: Core State Machine

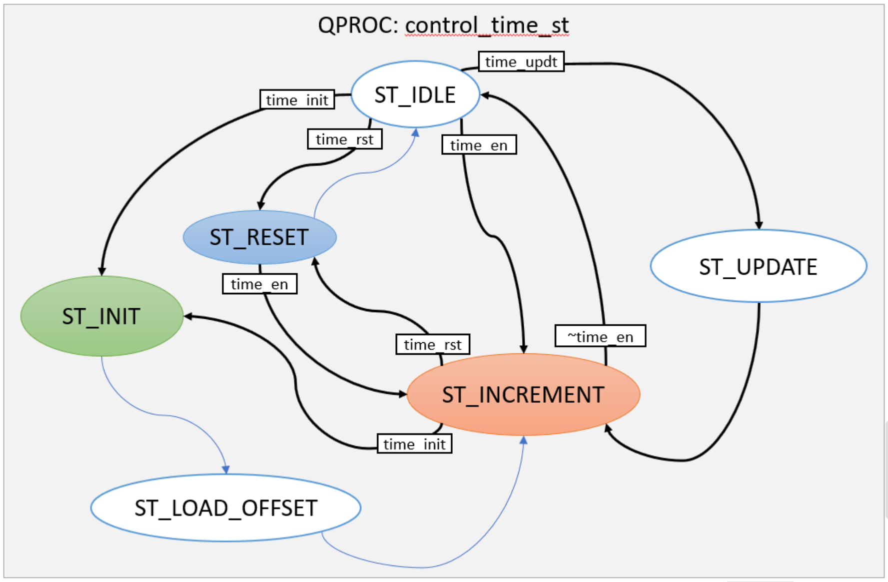
7.1.2 Time Control

TIME_RST > When the time_rst event is generated the time counter and the reference time are reset (*time_abs* and *ref_time*). And after time counter start counting.

TIME_INIT > When the time_init event is generated the time counter and the reference time are reset (*time_abs* and *ref_time*). And after time counter start counting.

TIME_UPDATE > This event changes the state of the core from C_STOP to C_RUN.

When the core is reset, the Program Counter (PC) is set to address 0, all the FIFOS are flushed, all the peripheral status bits (new data and ready) are reset , the flags are set to 0, and all the sreg, dreg and wreg are clear. The LFSR is not cleared.



When the processor makes a PORT_WR the data is pushed into the FIFO. It takes 2 clock cycles to the FIFO to Update the value to the OUTPUT. One more clock cycle for the Comparator to compare the value of the TIME in the FIFO with the tima_abs. The the POP signal is set and with one register for speed we have the ouput in the PORT, 5 clocks after was written by the qick_processor.

7.1.3 Configurable Parameters

- IN_PORT_QTY > Data Port In quantity (Up to 8)
- OUT_DPORT_QTY > Data Port Out quantity (Up to 4)
- OUT_WPORT_QTY > Wave Port Out quantity (Up to 16)
- PMEM_AW: Program Memory Address width. This parameter defines the Program Memory Size.
- DMEM_AW: Data Memory Address width. This parameter defines the Data Memory Size.
- WMEM_AW: Wave Memory Address width. This parameter defines the Wave Memory Size.
- REG_AW > Data Registers (Up to 32)

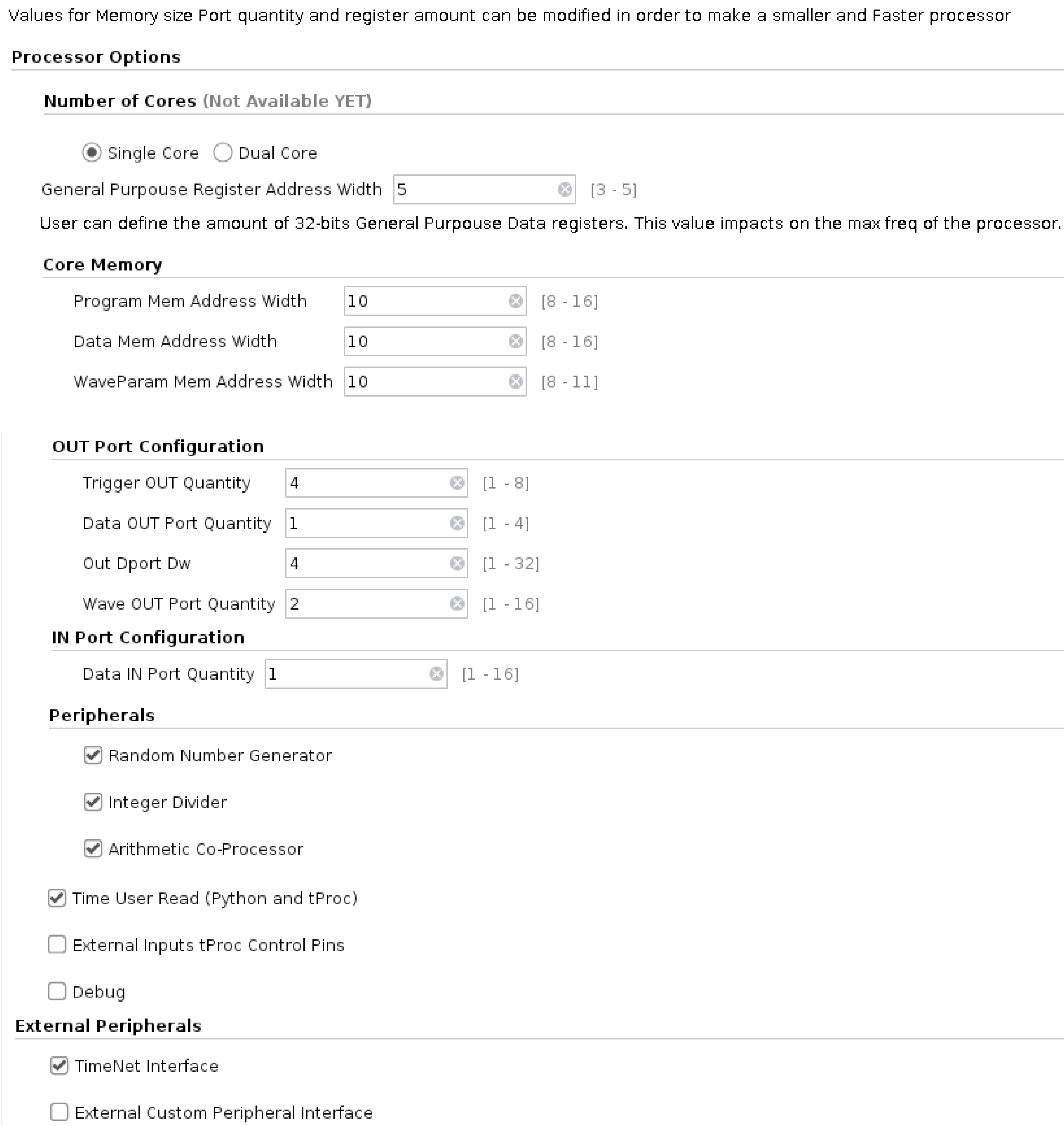


Figure 14: Configuration of qick_processor

7.2 PIPELINE STAGES

Fetch – Decode – ReadReg - Execute_First - Execute_Second-Write_Back

Fecth > Get the Instruction From Memory

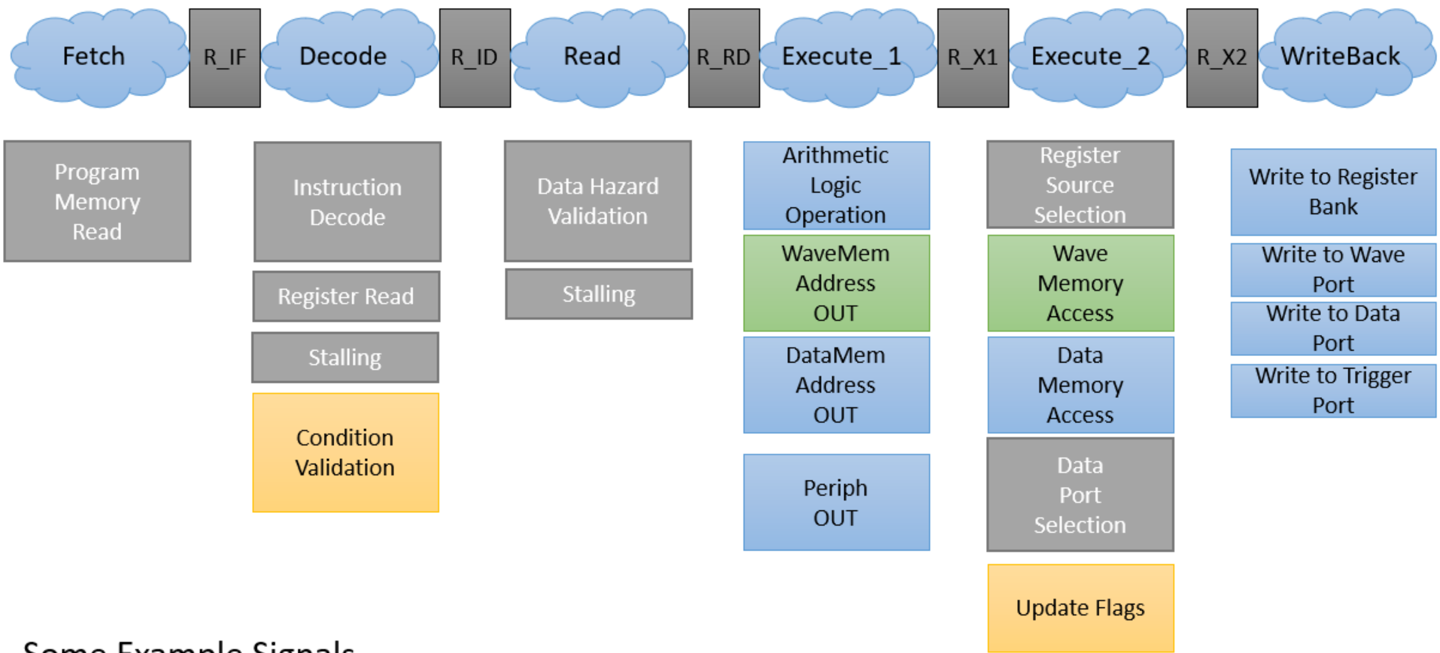
Decode > Generates all the Control signals from the OP_CODE and Access the Register Bank and Get all the Data signals

Read > Get the Data from the Pipeline if was previously processed.

Execute_First > Calculate the ALU Operation, Calculates the DataMem and Wmem Address. Update the ALU FLAGS

Execute_Second > Read/Write Data Memory or Wave Memory

Write_Reg > Write the Final Result to the register Bank or to the Output Port.



Some Example Signals

if_op_code if_op_data	id_reg id_ctrl id_imm_dt id_rs_D_addr id_imm_addr id_rs_A_addr	rd_reg rd_ctrl r_id_imm_dt Hzd/reg_D_nxt r_id_imm_addr Hzd/reg_A_nxt	x1_reg x1_ctrl r_rd_imm_dt reg_D_fwd_dt r_rd_imm_addr reg_A_fwd_dtr x1_mem_w_dt x1_mem_addr	x2_reg x2_ctrl x2_reg_w_dt r_x1_port_w_addr x2_port_w_dt

7.3 Clock Information

The block has several clocks.

T_clk > Clock used to count time. This should be the FASTEST clock.

C_clk > Clock used to run the processor. Period equal or greater than t_clk (Frequency Should be equal or slower than t_clk)

PS_clk > Clock used to communicate with the PS. Is the clock of the AXI.

7.4 Signals Names and Convention

7.5 FIFO SELECTIONS

The depth of the WFIFO, DFIFO and TFIFO can be selected. The table shows the utilization depending on the amount of bits used to address the FIFO.

Bits	RAM36	RAM18	CLB-LUT	CLB-FF	Distributed RAM
6	8	0	6587	6731	1168
7	20	0	5852	5034	352
8	23	0	5447	4782	0
9	23	0	5509	4838	0
10	37	2	5538	4895	0

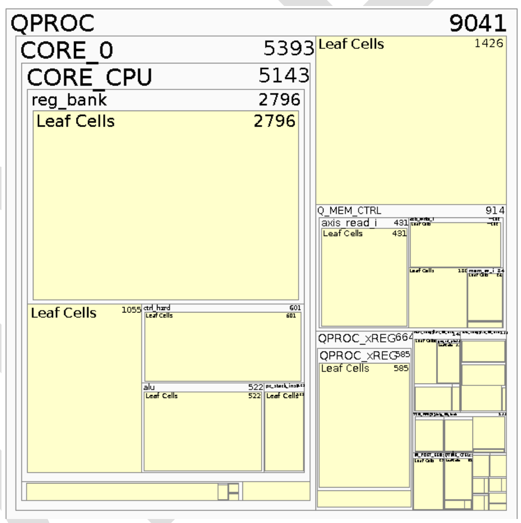
Smaller Version >

No external Signals, No peripherals

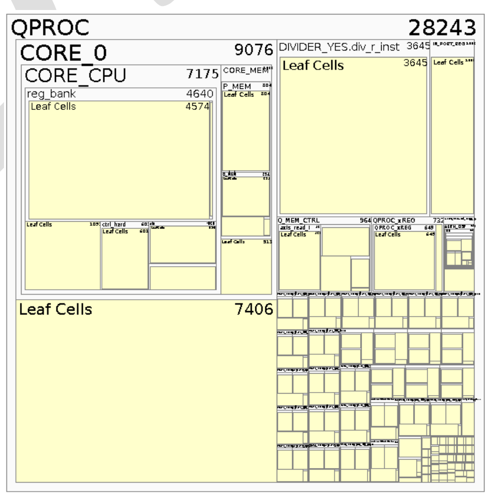
FIFO_DEPTH = 8, PMEM_AW = 8, DMEM_AW = 8, WMEM_AW = 8, REG_AW = 4,

IN_PORT_QTY = 1, OUT_TRIG_QTY = 1, OUT_DPORT_QTY = 1, OUT_DPORT_DW = 4, OUT_WPORT_QTY = 1

Parameter	Value
DEBUG	NO
PERIPH	NO
FIFO_DEPTH	8
PMEM_AW	8
DMEM_AW	8
WMEM_AW	8
REG_AW	4
IN_PORT_QTY	1
OUT_TRIG_QTY	1
OUT_DPORT_QTY	1
OUT_DPORT_DW	4
OUT_WPORT_QTY	1



Parameter	Value
DEBUG	REG
PERIPH	ALL
FIFO_DEPTH	9
PMEM_AW	8
DMEM_AW	8
WMEM_AW	8
REG_AW	5
IN_PORT_QTY	16
OUT_TRIG_QTY	8
OUT_DPORT_QTY	4
OUT_DPORT_DW	32
OUT_WPORT_QTY	16



Bigger Version > All external Signals, All peripherals

FIFO_DEPTH = 9, PMEM_AW = 16, DMEM_AW = 16, WMEM_AW = 11, REG_AW = 5,

IN_PORT_QTY = 16, OUT_TRIG_QTY = 8, OUT_DPORT_QTY = 4, OUT_DPORT_DW = 32, OUT_WPORT_QTY = 16

QICK Version > All external Signals, All peripherals

FIFO_DEPTH = 9, PMEM_AW = 10, DMEM_AW = 10, WMEM_AW = 10, REG_AW = 5, IN_PORT_QTY = 4
OUT_TRIG_QTY = 4, OUT_DPORT_QTY = 1, OUT_DPORT_DW = 4, OUT_WPORT_QTY = 1

VERSION	RAM36	RAM18	DSP48	CLB-LUT	CLB-FF	CARRY8	F7MUXES	Distributed RAM	Logic	Net	Total
SMALLER	16	2	4	3938	3438	28	654	0	1.5	1.5	3
BIGGER	307	13	30	10636	11515	226	1434	0	3.7	1.5	5.2
QICK	37	0	14	6519	5997	227	920	0	1.7	1.5	3.2

8 DEBUGGING

PYTHON EXAMPLES:

```
print('Program FPGA')
soc = PfbSoc('./tproc.bit', ignore_version=True, init_clks=True)

print('Write Program Memory')
wr_buff = program_memory
soc.tproc.tproc_cfg       = 7
dma.sendchannel.transfer(wr_buff)
dma.sendchannel.wait()
soc.tproc.tproc_cfg       = 0

print('Reset tProc')
soc.tproc.tproc_ctrl      = 1 # RST
print('Start Processing')
```

```
soc.tproc.tproc_ctrl      = 16 # PLAY
print('Set External Condition')
soc.tproc.tproc_ctrl      = 1024 # COND set
print('Stop Processing')
soc.tproc.tproc_ctrl      = 2 # STOP

print('Read Data from tProc')
tproc_time_usr = soc.tproc.time_usr
print('TIME_USR: ', time_usr)
tproc_rand = soc.tproc.rand
print('RAND: ', tproc_rand)
tproc_d1 = soc.tproc.tproc_ext_dt1_o
print('TPROC_EXT_DT_O: ', tproc_d1)
tproc_d2 = soc.tproc.tproc_ext_dt2_o
print('TPROC_EXT_OP_O: ', tproc_d2)

print('Continue Processing')
soc.tproc.tproc_ctrl      = 16 # PLAY

print(Read Data Memory')
soc.tproc.mem_addr        = 0
soc.tproc.mem_len         = 522
soc.tproc.tproc_cfg       = 9
dma_acc_recv.transfer(rd_buff)
dma_acc_recv.wait()
soc.tproc.tproc_cfg       = 0
print(rd_buff)
```

The Block qick_processor has a Parameter clles DEBUG.

When this parameter is 0, there are no debug signals connecte.

The AXI registers t_status and t_debug are 0

Write (and read) to (from) memories from the PS interface can be done in two different ways. Using a 256-bit DMA controller or with an AXI command.

Using the DMA

The start logic of the transactions triggers on a rising edge of the tproc_cfg[0]:

TPROC_CFG								
Operation	VALUE	BANK		SRC	MEM		OP	S
		6	5	4	3	2	1	0
DMA Write PMEM	7					1	1	1
DMA Read PMEM	5					1		1
DMA Write DMEM	11				1		1	1
DMA Read DMEM	9				1			1
DMA Write WMEM	15				1	1	1	1
DMA Read WMEM	13				1	1		1
Single Write PMEM	23			1		1	1	1
Single Read PMEM	21			1		1		1
Single Write DMEM	27			1	1		1	1
Single Read DMEM	25			1	1			1
Single Write WMEM	31			1	1	1	1	1
Single Read WMEM	29			1	1	1		1

Table 20: Memory Operation Configuration

To Write 512 values to the Data Memory, starting in the address 127 , First the tProcessor should be configured and then, the DMA can be started as follow:

- `Write a 512 in AXI-register` **`mem_len`**
- `Write a 127  in AXI- register` **`mem_addr`**
- `Write a 11 (1011b)  in AXI- register` **`tproc_cfg -`** `DMA Write DMEM`
    - o **`tproc_cfg`**`[0] (START) should be 1, to start the transfer state machine.`
    - o **`tproc_cfg`**`[1] (Memory Operation)should be 1, to set a Write operation.`
    - o **`tproc_cfg`**`[3:2] (Memory Bank Selection)should be 10, to select Data Memory`
- `Make DMA Transfer`
- `Write a 0  in AXI- register` **`tproc_cfg`**

ASSEMBLER EXAMPLE:

```
soc = QickSoc('./zcu_216_net.bit')        > Load FPGA
print('Write Program Memory')
length = len(program_memory)              > Get memory Size
soc.tproc.mem_addr          = 0           > Set Start Address
soc.tproc.mem_len           = length      > Set Data Memory Lenght
soc.tproc.tproc_cfg         = 7           > Configure Memory BLock
dma.sendchannel.transfer(wr_buff)         > Make DMA Transfer
dma.sendchannel.wait()                    > Wait Until DMA ends
soc.tproc.tproc_cfg         = 0           > Write a 0  in AXI- register tproc_cfg
```

Single Data Read

To Read a Single Value from address 123 from the Wave Memory. First the qick_processor should be configured and then,the read can be done:

- `Write a 123 in AXI- register` **`mem_addr`**
- `Write a 29 (11101b)  in AXI- register` **`tproc_cfg`** `– Single Read WMEM`
- `Read AXI-Register` **`mem_dt_o`**
- `Write a 0 in AXI- register` **`tproc_cfg`**

ASSEMBLER EXAMPLE:

```
soc = QickSoc('./zcu_216_net.bit')
soc.tproc.mem_addr = 123
soc.tproc.tproc_cfg = 29

data = soc.tproc.mem_dt_o
soc.tproc.tproc_cfg = 0
```

```
> Load FPGA
> Write a 123  in AXI- register mem_addr
> Write a 29 (11101b)  in AXI- register
tproc_cfg – Single Read WMEM
> Read AXI-Register mem_dt_o
> Write a 0 in AXI- register tproc_cfg
```


TPROC CONTROL									
TASK	EXT	PYTHON		CORE	Time		Core		Description
Time Reset	Y	0	1	Y	RESET	RUN	RESET	PREVIOUS	Synchronize Time
Time Init	Y				INIT	RUN	RESET	PREVIOUS	
Time Update	Y	1	2	Y	UPDATE	RUN	RESET	PREVIOUS	
Proc Start	Y	2	4		RESET	RUN	RESET	RUN	Restart the processor
Proc Stop	Y	3	8			STOP		STOP	Stop ALL
Core Start	Y	4	16			PREVIOUS	RESET	RUN	For use with synchronized time in different boards. Start and stop only the cores.
Core Stop	Y	5	32			PREVIOUS		STOP	
Proc RST		6	64		RESET	RUN	RESET	STOP	Debug
Proc RUN		7	128			RUN		RUN	
Proc PAUSE		8	256			RUN		STOP	
Proc FREEZE		9	512			STOP		RUN	
Proc STEP		10	1024		INC	STOP	INC	STOP	
CORE_STEP		11	2048			PREVIOUS	INC	STOP	
TIME_STEP		12	4096		INC	STOP		PREVIOUS	

AXI_Register Bit				
BIT	TPROC_CTRL	TPROC_CFG	CORE_CFG	READ_SEL
0	TIME_RST	MEM_START	LFSR0_CFG	Source
1	TIME_UPDATE	MEM_OP		
2	PROC_START	MEM_TYPE (PMEM, DMEM, WMEM)	RFU	
3	PROC_STOP			
4	CORE_START	MEM_SOURCE		RFU
5	CORE_STOP	MEM_BANK (TPROC, CORE0, CORE1)		
6	PROC_RST			
7	PROC_RUN	RFU		
8	PROC_PAUSE			
9	PROC_FREEZE			

10	PROC_STEP	DISABLE NET_CTRL		
11	CORE_STEP	ENABLE IO_CTRL		
12	TIME_STEP	DISABLE FIFO_FULL_PAUSE		
13	FLAG_SET			
14	FLAG_CLR			

Table 21: AXI-Registers

DEBUG OPTIONS

- ***Processor Reset*** > Reset the time, reset the core. Time starts running and core is paused.
- ***Processor Run*** > Starts / Continue Core and Time.
- ***Processor Pause*** > Stops the execution of the current program, but time increments.
- ***Processor Freeze*** > Stops the time increment, but processor continues running.
- ***Time Step*** > Time increments ONE.
- ***Core Step*** > Core executes ONE instruction.
- ***Processor Step*** > Time and Core Step.

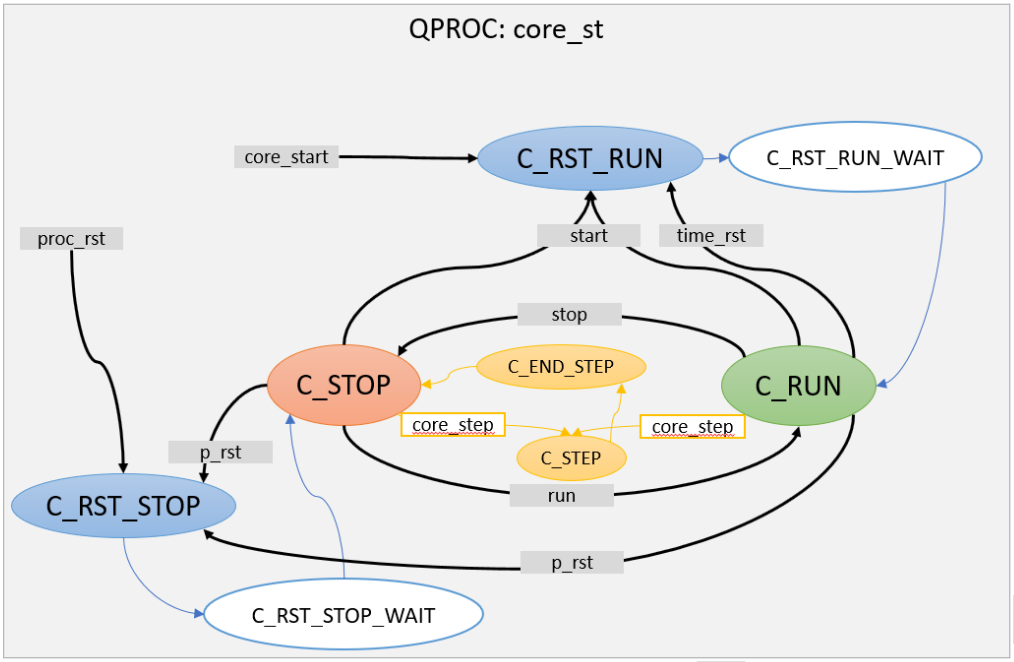
AXI-Register	Bit	Description
tproc_cfg	0	Start Memory Operation (1-Start) To make a new, Go to 0 first.
	1	Memory Operation selection (0-Read, 1-Write)
	3..2	Memory Bank Selection for Operation (01-Pmem , 10-Dmem , 11-Wmem)
	4	Memory Operation Data Source selection (0-AXIS, 1-REGISTERS (Single Read))
	6:5	Memory Bank Selection (Core0, Core1)
	9	Disable QNET Control (default 0: Yes Control from QNET)
	11	Enable IO Control (default 0: No control from IO)
	10	Debug (DISABLE FIFO_FULL_PAUSE)

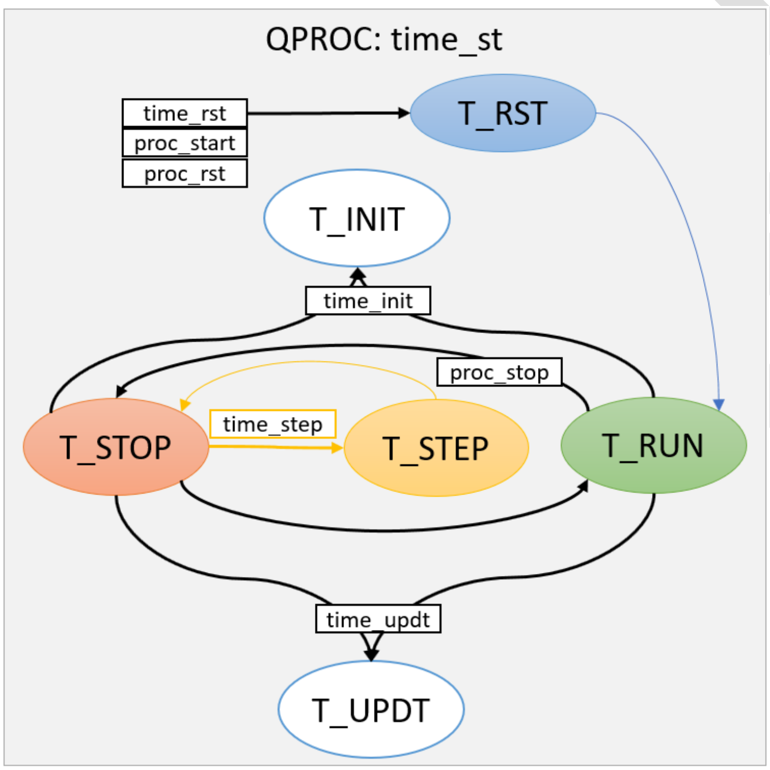
AXI_REG		
BIT	TPROC_STATUS	TPROC_DEBUG
0	core_st	fifo_time
1		
2		
3	core_en	
4	time_st	fifo_data
5		
6		
7	time_en	
8	int_flag_r	ref_time
9	ext_flag_r	
10	0	
11	flag_c0	
12	all_tfifo_empty	
13	all_dfifo_empty	
14	all_wfifo_empty	
15	all_fifo_empt	
16	all_tfifo_full	ext_mem_w_dt_o[7:0]
17	all_dfifo_full	
18	all_wfifo_full	
19	all_fifo_full	
20	tfifo_full	
21	dfifo_full	
22	wfifo_full	
23	fifo_ok	
24	ctrl_p_start	ext_mem_addr [7:0]
25	ctrl_p_stop	
26	ctrl_p_rst	
27	ctrl_p_run	
28	ctrl_p_pause	
29	ctrl_p_freeze	
30	aw_exec	
31	ar_exec	

DEBUGGING

The qick_processor is controlled and monitored from a python interface.
Possible Commands from Python.

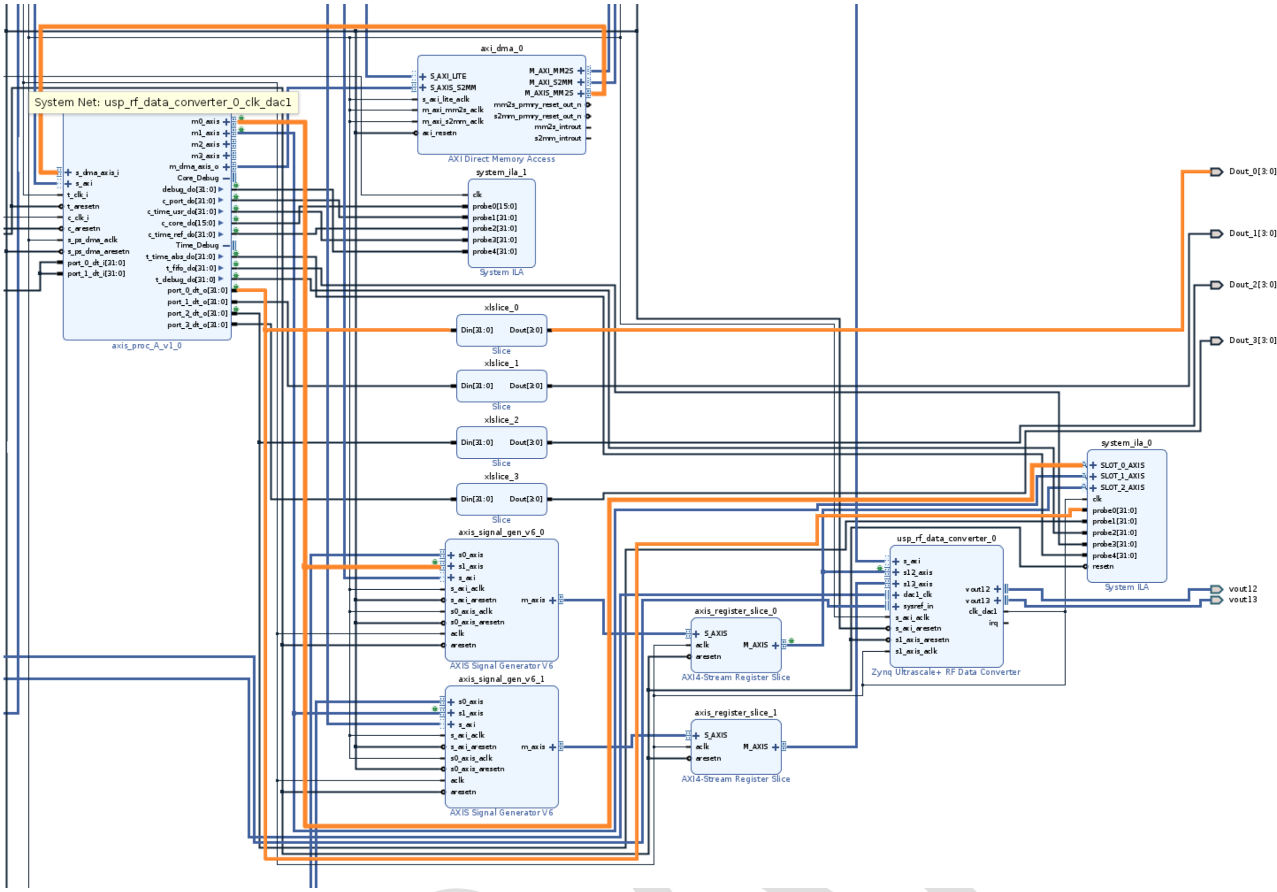
- ***Read / Write Program memory*** > Write or Read a program to/from the Program Memory
- ***Read / Write Data memory*** > Write or Read Data to/from the Data Memory
- ***Read / Write WaveParam memory*** > Write a list of Waveform to be used during the execution of the program in the WaveParam Memory.
- ***Time Reset, Init, Update*** > Time can be changed to reset to zero, initialized with any value or incremented a defined value.
- ***Core Reset*** > Reset the qick_processor. The Instruction pointer returns to Zero, all the general-purpose registers are cleared, and the FIFOs are flushed. (The AXI Registers keep their values)
- ***Play*** > Starts / Continue with the execution of the current program.
- ***Stop*** > Stops the execution of the current program. This command stops the Processor and stop the timing (time does not run). To continue running send **Play**
- ***Pause Core*** > Stops the execution of the current program, but time continues running on the tProcessor.
- ***Freeze Time*** > Stops the timing, but processor continues running.
- Debugging Commands: ***Processor Step, Time Step, Core Step, Read Status and Debug Signals***





TIMING DETAILS>

Time counter start 2 cycles after the Processor starts.
Timing Control Instructions must cross clock domain, so it takes 2 clock to go, and 2 to return to execute. So. DO NOT execute 2 TIME instruction with less than 5 clocks on the slowest Domain.


System Net: usp_rf_data_converter_0_clk_dac1
axi_dma_0
AXI Direct Memory Access
system_ila_1
System ILA
axis_proc_A_v1_0
xlslice_0
xlslice_1
xlslice_2
xlslice_3
Slice
axis_signal_gen_v6_0
axis_signal_gen_v6_1
AXIS Signal Generator V6
axis_register_slice_0
axis_register_slice_1
AXI4-Stream Register Slice
usp_rf_data_converter_0
Zynq Ultrascale+ RF Data Converter
system_ila_0
System ILA
Dout_0[3:0]
Dout_1[3:0]
Dout_2[3:0]
Dout_3[3:0]
vout12
vout13

TPROC_STATUS	31..30	T_MEM	{ar_exec, aw_exec}
	29..24		{ext_P_mem_en, ext_P_mem_we, ext_D_mem_en, ext_D_mem_we, ext_W_mem_en, ext_W_mem_we}
	12..20	T_CORE	{ fifo_data_empty[0], fifo_data_full[0], fifo_data_empty[1], fifo_data_full[1]}
	19..16		{ fifo_wave_empty[0], fifo_wave_full[0], fifo_wave_empty[1], fifo_wave_full[1] }
	15..12		{ pmem_en_o, dmem_we_o, wmem_we_o, port_we}
	11..8		{ 2'b00 , time_cond_r, ext_cond_r}
	7..4		{ 2'b00 , time_en, proc_en}
	3..0		{ 1'b0 , proc_st}
TPROC_DEBUG debug_do	15..8	T_PROC	c_time_ref_dt[7:0]
	7..4		fifo_data_time[0][3:0]
	3..0		{ fifo_ok, pmem_dt_i[71:69] }
	31..24	T_MEM	ext_mem_addr [7:0]
	23..16		pmem_addr_i[7:0]
c_core_do	21..24	T_CORE	{restart_i, stall, flush, id_flag_we, alu_fZ_r, alu_fS_r, x2_ctrl.port_we, x2_ctrl.port_re}
	23.16		{id_ctrl, id_cfg, id_br, id_wr, id_wm, id_wp, id_dreg_we, id_dmem_we }
	15..8		r_x1_alu_dt[7:0]
	7..4		reg_time[3:0]
	3..0		port_o.p_time[3:0]
c_port_do	31..16	T_PROC	fifo_data_in_r[15:0]
	15..0		fifo_time_in_r[15:0]
c_time_usr_do	31..0	T_PROC	c_time_usr
c_time_ref_do	31..0	T_PROC	c_time_ref_dt
t_time_abs_do	31..0	T_PROC	time_abs
t_debug_do	31..28	T_PROC	{ fifo_data_full[3], fifo_data_full[2], fifo_data_full[0], fifo_data_full[0] }
	27..24		{ fifo_data_empty[3],fifo_data_empty[2],fifo_data_empty[1],fifo_data_empty[0] }
	23..20		{ data_pop[3],data_pop[2],data_pop[1],data_pop[0]}
	19..16		{ fifo_data_push_r[3], fifo_data_push_r[2], fifo_data_push_r[1], fifo_data_push_r[0] }
	15..12		{ D_RESULT[3][47], D_RESULT[2][47], D_RESULT[1][47], D_RESULT[0][47] }
	11.. 8		{data_pop_prev[3],data_pop_prev[2],data_pop_prev[1],data_pop_prev[0] }
	7 .. 4		fifo_data_time[2][47:44]
	3 .. 0		time_abs[47:44]
t_fifo_do	31..28	T_PROC	fifo_data_dt[2][3:0]
	27..16		fifo_data_time[2][11:0]
	15..12		fifo_data_dt[0][3:0]
	11..0		fifo_data_time[0][11:0]

Hazards NOT Verified FOR DUAL INSTRUCTIONS

REG_WR r_wave wmem [&0] -ww
REG_WR r_wave wmem [&0] -ww
R_wave and wmem should keep equal. But the cycle to read is not stalled.

Clocks:
The block has four clocks.

- ps_clk_slow (100Mhz) Clock used for AXI communication with PS
- c_clk (350 MHz) Clock used in the tCore Processor.
- t_clk (500 MHz) Clock used to run Time, clock related with the DACs
- read_clk Clock used to read data, clock related with the ADCs

8.1 ADDITIONAL INFO TO BE ADDED AND TAKEN INTO ACCOUNT

- When Configuring the FPGA all the OUTS goes to 1 (around 55 ms)
- In this version Address can only came from a DREG
- When tproc reset (Flush all FIFOS and data in memory pipeline should be discarded. That is why address 0 stores a NOP, in the reset the address read is 0)
-

Operation	Description	Example
ADD	Register plus Immediate	`-op(r1 + #7)`
	Register Plus Register	`-op(s1 + r2)`
SUB	Register minus Immediate	`-op(r1 - #7)`
	Register minus Register	`-op(s1 - r2)`
AND	Arithmetic Shift Right Immediate Amount	`-op(r1 ASR #7)`
	Arithmetic Shift Right registered Amount	`-op(s1 ASR r2)`
ASR	Arithmetic Shift Right Immediate Amount	`-op(r1 ASR #7)`
	Arithmetic Shift Right registered Amount	`-op(s1 ASR r2)`

Command	Conditional Execution	Second Data Task	Second Wave Task	Second Port Task	TEST Task
NOP					
TEST					YES
REG_WR	YES				YES
DPORT_RD					
DMEM_WR	YES	YES			YES
DPORT_WR	YES	YES	YES		YES
REG_WR		*YES	YES	YES	YES
WMEM_WR		YES	YES	YES	YES
WPORT_WR		YES	YES		
JUMP	YES	YES		YES	YES
CALL		YES			YES
RET		YES			YES
TIME					
ARITH					
DIV					
COND					