tProcessor-64 and Signal Generator V4

This note describe the details of the 64-bit timed-processor (tProc). This block is a custom processor, which means it executes a program loaded into a memory, allowing to target a wide variety of algorithms by modifying the program. The feature that makes this a block somehow different from standard processors is the addition of timed-instructions that are executed at specific times. These instructions differ from standard instructions in which they are dispatched into a timed-instruction control queue that executes them in order and at the specified time. The user can use standard instructions to operate on registers, push/pop to stack and compute times, to make things happen at very specific times using the timed-instructions. This block is the 64-bit version of the tProc, which can be connected to the Signal Generator V4 that adds phase control and uses a 160-bit interface.

1 Block diagram and features

Figure 1 shows the block diagram of the tProcessor block.The tProcessor has a Main Control state machine that decodes the instructions coming from the external program memory (PM). This memory is reserved for program only, and the user cannot read or write from the processor into this space..

After reset or when the user provides a stop and start sequence, the processor will start reading the memory from address 0. Each memory location has a 64-bit number which encodes the instruction together with the necessary parameters for the execution. The start and stop sequence can be applied with internal registers or with an external physical pin. This pin is intended when multiple processor blocks are instantiated and

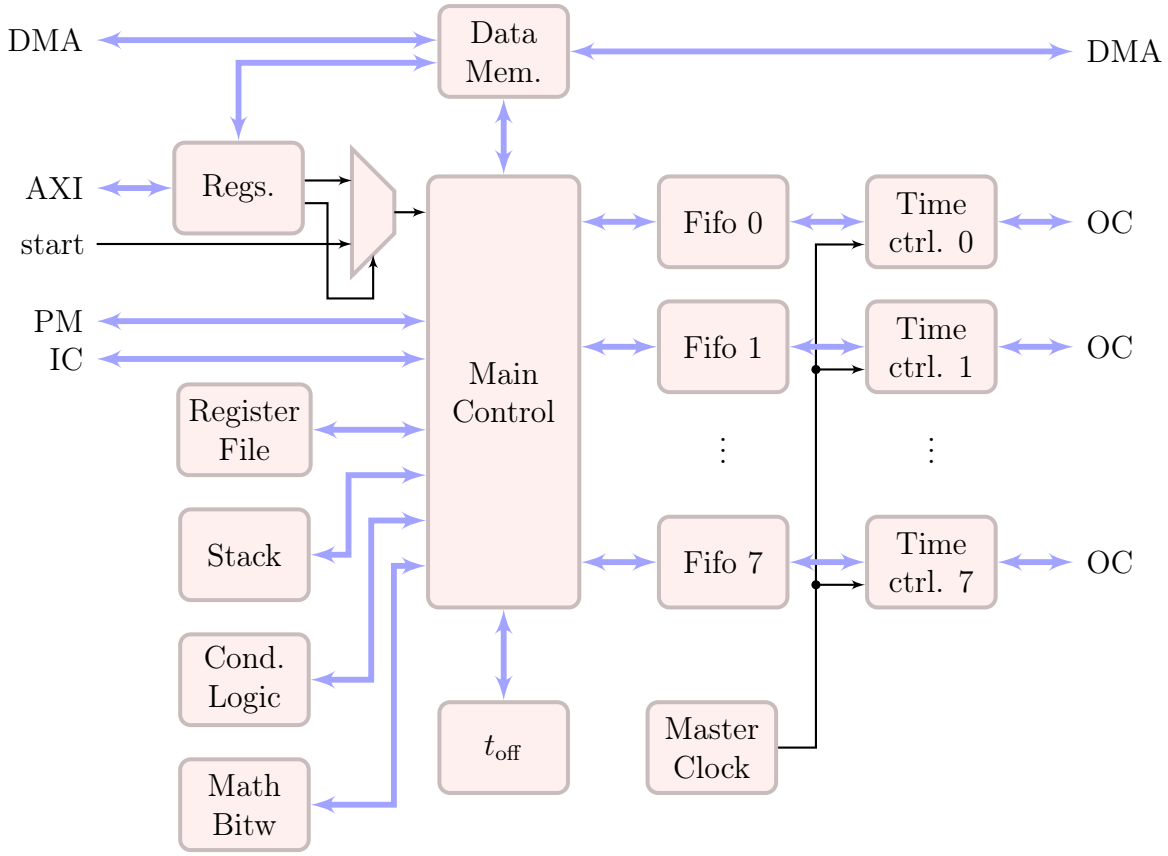

Figure 1: tProcessor block diagram with their main components.

they all need to start at the same time. Register 0 of the AXI address map is the source of the start, with a value of 0 for internal start and 1 for external start. Register 1 is the start register. A value of 1 will start the block. Once the program is completed it is necessary to reset this register to 0 and back to 1 in order to re-initialize the program execution. There are 4 extra registers that can be accessed, which are used for the internal data memory. These will be described later.

Not to confuse with the previously mentioned AXI registers is the Register File sub-block. Most of the instructions operate over registers, which are within this block. This implementation provides 8 pages, with 32 registers on each page. Registers are 32-bit. The register file provides the flexibility to read up to 7 registers and write 1 register on a single clock cycle. The user can initialize registers before time-critical instructions are issued to save computation time of the algorithm. Register number 0 on all four pages is hardwired to 0. This zero-valued register can be used to initialize registers or for comparison purposes.

A Stack block is also included with 256, 32-bit words. This stack can be accessed using push/pop instructions. It is up to the user to carefully handle the stack to avoid empty/full conditions. If either condition is reached, the processor will jump into a special error location and will need to be re-initialized. The stack can be used as a temporary memory, and becomes useful to implement nested loops as it allows using a single register as the loop register, with extra push/pop instructions in order to save or recall the loop value. The stack can also be used to move registers from page to page, provided there is no direct instruction for that matter.

The block of Cond. Logic implements the conditional logic that is used by the appropriate instruction. The inclusion of this logic ensures 1 clock cycle latency for the comparison, and also allows to expand to more complex conditions by expanding this sub-block. The instruction that makes use of this block will jump if the output flag is set to 1, which is the true condition. Other arithmetic and bit-wise operations are implemented by the Math/Bitw block. The Math part includes addition, substraction and product. Bit-wise operations include the standard and, or, exclusive or, not, left and right shift. There are immediate and register-like versions of both.

Timed-instructions make this processor a distinct block from standard processors. Timed instructions make use of the Master Clock and t_{off} register. The master clock is a counter that starts counting when the software starts at address 0, and never stops. This is a 48-bit counter, which clocked at the maximum FPGA speed of 500 Mhz gives roughly 156 hs of counting. This could still be expanded to use more bits in the future. It can be seen from the figure that the master clock is an input to the Time Ctrl. logic. The other piece that timed-instructions need is the time offset t_{off} register. This internal register is the time reference that timed-instructions refer to. As an example, let's say the processor starts executing and a timed-instruction is scheduled to happen at time 150. This means that when the master clock reaches 150, that particular instruction will be executed. In the case of looping through an algorithm, having to specify absolute times would be rather difficult, and the instruction itself would need to reserve 48-bit for this field. The processor includes special instructions to synchronize the time offset register.

Back to our example, let's imagine the algorithm needs to loop 10 times executing the timed-instruction at time 150 for the first time, at time $T + 150$ the second time, at $2T + 150$ for the third time, and so on. Using the sync mechanism, the user would need to provide the timed-instruction with a specification of 150 and then a sync instruction with T as the value. The first execution of the loop will have a value of 0 for t_{off}, and the timed-instruction will be scheduled to happen at 150 units. After the first sync instruction is provided, the internal t_{off} register will be set to T. On the second pass of the loop, the timed-instruction will be scheduled at time t_{off}+150, resulting in the absolute time of $T + 150$. This process continues until the loop is completed. In summary, when the processor decodes a timed-instruction it will compute the absolute time with the offset register before dispatching it into the timed-instruction control queue of that particular channel. This simple mechanism makes it easy to specify timing sequences, and also allows lowering the number of bits that need to be used in order to specify the time on timed-instructions. Additionally, because the absolute time of instructions is computed before dispatching them into the time-queue, the processor can continue decoding and looking-ahead of the program as the synchronization mechanism is nothing but defining a time reference for instructions.

Now the timed-instruction mechanism has been explained, going back to the diagram of Fig. 1, it can be seen the tProcessor provides 8 output channels (OC), implemented as axi-stream master interfaces. Each channel has its own fifo and associated time-control. This allows to execute instructions at the same absolute time in different channels. This may be useful in some special cases where events need to be either overlapped or executed at the same time for a specific experiment.

The processor can also read a single value from the provided input channels (IC). This could be used for example to read the result of a measurement, and take a decision based on that result. The read instruction, used for reading the value of this port, is implemented as non-blocking, which means that if the interface does not have the valid signal asserted, the instruction will still read the value and return to the main program. There is a mechanism that receives the data from the interface using the valid/ready

handshake and updates the value of an internal register. The user should then make sure the value is taken from the interface after the time it occurred. The instruction was designed as non-blocking to ensure the correct timing flow throughout the program executing. A blocking procedure will break the time structure of the algorithm.

The Data Mem. block is an internal data memory, independent from the program memory of the processor. This internal data memory can be accessed in different ways, and should be used to load data into the processor, or to read data from it. Memory can be accessed from the exterior using different methods, which are single-read/write, and DMA-like read/write. The processor can use the provided memory read/write instruction, either immediate or register based. The details of the operation of this data memory block will be introduced on a later section.

2 Instruction types

There are three types of instructions: `I-type`, `J-type` and `R-type`. The definition of these types allow to define the underlaying binary format of the instruction, together with the different bit fields. In the following lines the format of these instructions will be defined.

`I-type`

These instructions are called immediate type. The name indicates that there is a value, called the immediate value, that is used on the execution of the instruction and is specified as part of the instruction itself. This type of instruction is useful when a fixed value wants to be added to a register, or when a fixed amount of time needs to be used to synchronize the internal time offset. The `I-type` instructions implemented are:

- `pushi`: push value into the stack.
- `popi`: pop value from the stack.
- `mathi`: add, substract or multiply value to resiter.

- `seti`: set output port.
- `synci`: synchronize internal time offset.
- `waiti`: wait until master clock reaches specified time.
- `bitwi`: perform bit-wise operation on register.
- `memri`: read data memory.
- `memwi`: write data memory.
- `regwi`: write value into register.

Note that all immediate instructions have the "i" added at the end to make explicit reference they are immediate instructions. This help differentiating register-based instructions with similar notation and behavior. The format of the instruction is as follows:

63 : 56	55 : 53	52 : 50	49 : 46	45 : 41	40 : 36	35 : 31	30 : 0
`opcode`	`page`	`ch`	`oper`	`ra`	`rb`	`rc`	`imm`

As it can be seen from the instruction, there are 8 bits for the opcode. This is the same for all instructions. The page is 3 bits and it allows specifying up to 8 register pages. Registers are addressed using 5 bits, which allows to specify up to 32 registers per page. Bits [52 : 50] specify the output channel. It will be clear what the meaning of the channel is when describing the specifics of the instructions. The field `oper` is used in some instruction to specify the operation, like in the mathematical instructions. These instructions can use two source registers, one destination register and the immediate value. There is only one caveat at the moment. Immediate values are specified using only 31 bits instead of 32. This is due to legacy field description. This issue will be fix in an upcoming version.

`J-type`

This instruction is of jump type, which means the program counter of the processor may jump to an address that is not in the normal sequence of operation. In general, jump

instructions may or may not cause the processor to jump, depending on the result of the instruction. If the processor needs to jump, the address pointing to the memory is loaded with the new address. If the instruction does not indicate a jump needs to be applied, the processor skips to the next instruction and continues with the sequential execution. The `J-type` instructions that are implemented are:

- `loopnz`: jump and decrement if register is not zero.
- `condj`: jump if condition is true.
- `end`: finish program execution and jump to end state.

The `end` instruction is not actually a jump instruction, because it does not make the processor to read from the memory at a particular address. It causes the processor to get into the end state, which needs a stop and start cycle to execute a new program or repeat the one loaded into the memory. Bits [52 : 50] and [30 : 16] are not used. The format of `J-type` instructions is as follows:

63 : 56	55 : 53	49 : 46	45 : 41	40 : 36	35 : 31	15 : 0
`opcode`	`page`	`oper`	`ra`	`rb`	`rc`	`addr`

`R-type`

These instructions are register-based. Even if the other instructions operate with registers most of the times, `R-type` instructions operate always using registers and not immediate values. The purpose of these instructions is to allow operating on up 7 registers as source, plus an extra register as destination. There are not any instruction implemented yet which uses all those registers at once, but it is possible given the structure of this type of instruction. The instructions implemented of this type are:

- `math`: addition, subtraction or product of two registers, writing write the result on a third register.
- `set`: set output port using 5 registers as source, and a sixth register for time.

- `sync`: synchronize the internal time offset.
- `read`: read input port value into register.
- `wait`: wait until master clock reaches time specified on register.
- `bitw`: apply bit-wise operation on registers.
- `memr`: read data memory.
- `memw`: write data memory.

As mentioned before, these instructions perform a similar operation to their immediate counterparts, using registers instead of immediate value. For memory operations, the user could use a register to access the data memory with a loop variable, instead of a fixed immediate value. As an example of multiple register use, it can be seen that `set` instruction operates with 6 registers as source. This allows to output 160 bits with a binary word for the instruction that is 64 bits. This would not be possible using immediate instructions. The format of this type of instruction is as follows:

63 : 56	55 : 53	52 : 50	49 : 46	45 : 41	40 : 36	35 : 31	30 : 26	25 : 21	20 : 16	15 : 11	10 : 6
opcode	page	ch	oper	ra	rb	rc	rd	re	rf	rg	rh

3 Instruction set

Instructions are specified using assembly language. This language was developed to ease the writing of simple algorithms. What matters is the underlaying binary code which is loaded into the program memory of the tProcessor. In the upcoming pages, the assembly code, together with the description of the instruction and the binary machine code are going to be detailed. The user could use any other software mechanism to create the binary code in order to re-define the input language.

I-type

`pushi p, $ra, $rb, imm:`

This instruction will push the contents of register `$ra` into the stack, and load register `$rb` with the value specified by `imm`. Both registers are on page `p`, and can be the same to reuse the register. If the stack is full, this instruction will make the processor to jump into an error state.

63 : 56	55 : 53	52 : 50	49 : 46	45 : 41	40 : 36	35 : 31	30 : 0
00010000	page	xxx	xxx	rb	ra	xx	imm

Example: `pushi 0, $1, $1, 100`, push the content of register 1 on page 0 into the stack, and load 100 on the same register.

`popi p, $r:`

This instruction will pop the contents from the top of the stack into the register `$r` of page `p`. If the stack is empty, this instruction will make the processor to jump into an error state. Source register and immediate fields are not used and could be set to any value.

63 : 56	55 : 53	52 : 50	49 : 46	45 : 41	40 : 36	35 : 31	30 : 0
00010001	page	xxx	xxx	r	xx	xx	xx

Example: `popi, 1, $3`, pop the content of the stack into register 3 on page 1.

`mathi p, $ra, $rb oper imm:`

This instruction will apply the specified operation on register `$rb` and the value `imm`, and write the result into register `$ra`. The `imm` value is 32-bit signed, which allows implementing signed arithmetic. Due to the fact the immediate field is defined over 31 bits, it is internally sign-extended to get 32-bit before operating with it. Both registers are on page `p`. Registers `$ra` and `$rb` can be the same, to allow updating the value. There are three operations implemented so far:

- +, adds the contents of the register and the immediate value, is coded as 1000.

- $-$, substracts the immediate value from the register, coded as 1001.
- $*$, product of register and immediate, coded as 1010.

It's important to say that for the case of the product, the output is still coded as 32-bit signed number. The actual product is implemented using the lower 16 bits of the operands to create the output over 32-bit.

63 : 56	55 : 53	52 : 50	49 : 46	45 : 41	40 : 36	35 : 31	30 : 0
00010010	page	xx	oper	ra	rb	xx	imm

Example: `mathi 2, $2, $1 * 3`, compute the product of register number 1 on page 2, and the fixed number 3, and write the result into register number 2 on page 2.

`seti ch, p, $r, imm:`

This instruction will set the value specified by register `$r` on page `p`, to be the output of channel `ch` at time `imm`. As any timed-instruction, the processor will dispatch it to the timed-control queue to schedule it to happen in the future using the master clock as reference. Output channels of the tProcessor are 160-bit and this instruction uses one register, which is 32-bit. The output port will be zero-padded to complete 160-bit and the register value will set the lower 32 bits. The actual implementation of this instruction act as a non-blocking write to the output master axi-stream interface. This means the block won't wait until the corresponding ready signal is asserted and will act as if a "always ready" slave was connected. This behavior could easily be changed but could compromise timed-instructions in general, as the block would be blocked until the transaction is completed.

63 : 56	55 : 53	52 : 50	49 : 46	45 : 41	40 : 36	35 : 31	30 : 0
00010011	page	ch	xx	xx	r	xx	imm

Example: `seti 3, 0, $8, 65`, set channel 3 to the value specified by register 8 on page 0, at time 65.

`synci imm:`

This instruction is used to synchronize the internal time offset register to the value specified by `imm`. At the beginning of the execution of the program, the time offset is set to zero and any timed-instruction is referenced to 0. When a `synci` instruction is provided, the internal time offset is added the time value indicated by the instruction. From that point on, whenever a timed-instruction is decoded, the tProc will compute the absolute time using this internal time offset. This mechanism makes it very easy to specify timed-instructions relative to the last synchronization. Page, channel, operation, source and destination register fields are not used and could be set to any value.

63 : 56	55 : 53	52 : 50	49 : 46	45 : 41	40 : 36	35 : 31	30 : 0
00010100	xx	xx	xx	xx	xx	xx	imm

Example: `synci 500`, synchronize internal time offset to its previous value plus 500.

`waiti ch, imm:`

This instruction should be used when the algorithm needs to wait up to a certain time before proceeding. This could be useful, as an example, to read the value of the input port after a specified time. The instruction will wait until time `imm` on channel `ch` is reached. It is important to note that, even if the channel is selected, the decoding of instructions will stop until the master clock reaches the specified time. Instructions cannot be dispatched to other channels. In short, the user could use any of the available 8 channels to execute the `waiti` instruction. A side effect of this instruction is that the internal timed-control queue is empty on the channel after this instruction.

63 : 56	55 : 53	52 : 50	49 : 46	45 : 41	40 : 36	35 : 31	30 : 0
00010101	xx	ch	xx	xx	xx	xx	imm

Example: `waiti 2, 500`, wait until time 500 is reached on channel 2.

`bitwi p, $ra, $rb oper imm:`

This instruction will apply the specified operation between register `rb` and the `imm` value, and write the result back into register `ra`. The operations implemented are:

- 0000: `$rb & imm`, is the and operation.
- 0001: `$rb | imm`, is the or operation.
- 0010: `$rb ^ imm`, is the exclusive or operation.
- 0011: `~ imm`, is the not operation.
- 0100: `$rb « imm`, is the left shift operation.
- 0101: `$rb » imm`, is the right shift operation.

Note that the not operation has a slightly different syntax, and it does not use register `rb`.

63 : 56	55 : 53	52 : 50	49 : 46	45 : 41	40 : 36	35 : 31	30 : 0
00010110	page	xx	oper	ra	rb	xx	imm

Example: `bitwi 3, $4, $4 « 4`, perform a left shift of 4 positions over register 4 on page 3, and write the result into the same register.

`memri p, $r, imm:`

Read the data memory at the address specified by `imm` and write the result into register `r` on page `p`. Data memory can be pre-loaded using the methods provided, which will be detailed on a later section. Addressing of the data memory is sample-based, which means address 0 will read a 32-bit value, address 1 will read the next 32-bit value, and so on.

63 : 56	55 : 53	52 : 50	49 : 46	45 : 41	40 : 36	35 : 31	30 : 0
00010111	page	xx	xx	r	xx	xx	imm

Example: `memri 1, $3, 23`, read data memory at address 23, and write the value into register 3 of page 1.

`memwi p, $r, imm:`

Write register `r` of page `p` into data memory at address `imm`. This instruction should be used when the memory location is fixed and known in advance. For loops and variable-indexing address, the user may use the register-based instructions for memory addressing.

63 : 56	55 : 53	52 : 50	49 : 46	45 : 41	40 : 36	35 : 31	30 : 0
00011000	page	xx	xx	xx	xx	r	imm

Example: `memwi 2, $2, 11`, write contents of register 2 on page 2, on memory at address number 11.

`regwi p, $r, imm:`

Write the value specified by `imm` in register `r` on page `p`. This is instruction is widely used at initialization stages to set initial values on registers. As an alternative, memory read instructions can also be used to initialize registers from previously user-defined values which can be dynamically modified.

63 : 56	55 : 53	52 : 50	49 : 46	45 : 41	40 : 36	35 : 31	30 : 0
00011001	page	xx	xx	r	xx	xx	imm

Example: `regwi 0, $4, -38`, write the value -38 into register 4, page 0.

J-type

`loopnz p, $r, @label:`

This instruction will decrement by one the value of register `$r` on page `p`, and jump to the address indicated by `label`. If the register is zero when the instruction is executed, it is not decremented and the instruction skips to the next line of code without implementing the jump. The address in calculated by the compiler, as the user will normally indicate a label that points to an instruction in the memory. An immediate value could also be used. Note that as source and destination registers are the same, this value is repeated in the binary code.

63 : 56	55 : 53	49 : 46	45 : 41	40 : 36	35 : 31	15 : 0
00110000	page	1000	r	r	xx	addr

Example: `loopnz 1, $2, @LOOP`, jump to address on symbol `LOOP` if register 2 on page 1 is not zero, and decrement the register.

`condj p, $ra op $rb, @label:`

This instruction will jump to the address indicated by `label` if the condition is true. There are 6 operations implemented: >, >=, <, <=, == and ! = . Registers `$ra` and `$rb` are on page `p`. If the condition is false, the instruction will skip to the next instruction in the program. The typical if-elsif-else structure can be easily implemented chaining few `condj` instructions.

63 : 56	55 : 53	49 : 46	45 : 41	40 : 36	35 : 31	15 : 0
00110001	page	oper	xx	ra	rb	addr

The `op` field is coded as follows:

- > means greater than and is coded as 0000.
- >= means greater or equal and is coded as 0001.
- < means smaller than and is coded as 0010.
- <= means smaller or equal and is coded as 0011.
- == means equal and is coded as 0100.
- ! = means different and is coded as 0101.

Example: `condj 2, $4 != $5, @DIFF`, jump to address on symbol `DIFF` if register 4 on page 2 is different from register 5 on page 2.

`end:`

This instruction will end the program execution. The processor state machine will jump to the end state. To execute again, the user should send a stop followed by a start. All

fields but the `opcode` are not used and could be set to any value.

63 : 56	55 : 53	49 : 46	45 : 41	40 : 36	35 : 31	15 : 0
00111111	xx	xx	xx	xx	xx	xx

R-type

`math p, $ra, $rb oper $rc:`

Apply the specified operation to the contents of registers `$rb` and `$rc` and write the result on register `$ra`. Registers belong to page `p`. All three registers can be the same, which allows to update the value over the same register. Operations are coded the same way as for `mathi` operation. Addition, substraction and product are implemented.

63 : 56	55 : 53	52 : 50	49 : 46	45 : 41	40 : 36	35 : 31	30 : 26	25 : 21	20 : 16	15 : 11	10 : 6
01010000	page	xx	oper	ra	rb	rc	xx	xx	xx	xx	xx

Example: `math 0, $3, $3 + $4`, add the contents of register 3 and 4, and write the result back into register 3, all on page 0.

`set ch, p, $ra, $rb, $rc, $rd, $re, $rt:`

This instruction will set the value specified by registers `$ra`, `$rb`, `$rc`, `$rd` and `$re` on page `p`, to the output channel `ch` at the time specified by register `$rt`. Registers are 16-bits, and the lower bits are those of `$ra`, then it follows `$rb`, `$rc` and finally `$re`.

63 : 56	55 : 53	52 : 50	49 : 46	45 : 41	40 : 36	35 : 31	30 : 26	25 : 21	20 : 16	15 : 11	10 : 6
01010001	page	ch	oper	xx	ra	rt	rb	rc	rd	re	xx

Example: `set 0, 1, $1, $2, $3, $4, $5, $6`, set the value on registers 5, 4, 3, 2 and 1 to the output channel 0, at time on register 6. All registers are on page 1.

`sync p, $r:`

Synchronize internal time offset to the value specified by register `$r` on page `p`. This instruction has the same effect that `synci`, but time is specified with a register. Most of the fields of the instruction are unused and could be set to any value.

63 : 56	55 : 53	52 : 50	49 : 46	45 : 41	40 : 36	35 : 31	30 : 26	25 : 21	20 : 16	15 : 11	10 : 6
01010010	page	xx	xx	xx	xx	r	xx	xx	xx	xx	xx

Example: `sync 2, $3`, synchronize internal time offset to its previous value plus the one specified by register 3 on page 2.

`read p, $r:`

Read external port value into register `$r` on page `p`. This instruction could be used in conjunction with `wait` variants and `condj` to read a result at a specific time and jump according to the condition. The value is read from a axi-stream slave port. However, it is implemented as a non-blocking read to avoid dead-locking the processor until input data is valid. The input axi-stream slave is connected to a specific logic that implements the valid/ready handshake and sets a value when new data is received. This value can be read using this instruction, without blocking the processor. The user may issue a wait instruction to be sure data is received and execute the reading after that point.

63 : 56	55 : 53	52 : 50	49 : 46	45 : 41	40 : 36	35 : 31	30 : 26	25 : 21	20 : 16	15 : 11	10 : 6
01010011	page	xx	xx	r	xx	xx	xx	xx	xx	xx	xx

Example: `read 0, $1`, read the value of the input port to register 1 on page 0.

`wait ch, p, $r:`

Similar to `waiti`, the instruction will wait until time specified by register `$r` on page `p` has arrived on channel `ch`. All upcoming instruction decoding is suspended while waiting. This instruction should be used only when an external event needs to happen before continuing executing instruction. As an example, reading the result of an external device

and branching accordingly.

63 : 56	55 : 53	52 : 50	49 : 46	45 : 41	40 : 36	35 : 31	30 : 26	25 : 21	20 : 16	15 : 11	10 : 6
01010100	page	ch	oper	xx	xx	r	rd	xx	xx	xx	xx

Example: `wait 3, 1, $2`, wait until time specified by register 2 on page 1 is reached on channel 3.

`bitw p, $ra, $rb oper $rc:`

Apply the specified bit-wise operation over registers `rb` and `rc` and write the result back to register `ra`. Operations are coded the exact same way than for the `bitwi` operation.

63 : 56	55 : 53	52 : 50	49 : 46	45 : 41	40 : 36	35 : 31	30 : 26	25 : 21	20 : 16	15 : 11	10 : 6
01010101	page	xx	oper	ra	rb	rc	xx	xx	xx	xx	xx

Example: `bitw 3, $3, $3 & $4`, perform the bit-wise and operation between registers 3 and 4, and write the result back into register 3. Registers are all on page 3.

`memr p, $ra, $rb:`

Read data memory at address pointed by register `rb` and write the value into register `ra`. Registers are on page `p`. This instruction is useful when the address needs to be controlled on a loop, instead of accessing a fixed location as in the immediate counterpart.

63 : 56	55 : 53	52 : 50	49 : 46	45 : 41	40 : 36	35 : 31	30 : 26	25 : 21	20 : 16	15 : 11	10 : 6
01010110	page	xx	xx	ra	rb	xx	xx	xx	xx	xx	xx

Example: `memr 3, $5, $2`, read data memory at address pointed by register 2 and write the value into register 5, both on page 3.

`memw p, $ra, $rb:`

Write the contents of register `ra` into the data memory at the address pointed by register `rb`. As in the previous case, this instruction could be useful when trying to loop through

memory locations.

63 : 56	55 : 53	52 : 50	49 : 46	45 : 41	40 : 36	35 : 31	30 : 26	25 : 21	20 : 16	15 : 11	10 : 6
01010111	page	xx	xx	xx	rb	ra	xx	xx	xx	xx	xx

Example: `memw 1, $2, $13`, write the value of register 2 into the data memory at address pointed by register 13. Registers are both on page 1.

4 Signal Generator V2

Output channels of the tProc can be connected to Signal Generator V2 block. This block has a 80-bit interface, which allows "pushing" waveforms into it to reproduce with configuration parameters. The block includes a memory and a DDS block. The memory section can be used to upload an arbitrary shape. The DDS is a complex cosine/sine generator block, whose frequency can be configured using the provided interface. The block allows the user to select four different outputs: table, dds, product and zero-value. Table is real and DDS is complex. The output of this block is complex but the imaginary part is set to zero when table output is selected. Figure 2 shows a simplified block diagram of this block.

The table section is loaded using the `s0_axis` interface with a support DMA block. The user can specify the address of the first sample using the corresponding axi register, to allow uploading several waveforms into the internal memory. It is also necessary to enable writes into the memory before sending the samples in. This serves to protect agains unexpected writes. 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 `s1_axis` interface. This configuration interface is 80-bit and allows to push waveforms into the internal fifo or queue. 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. The meaning of

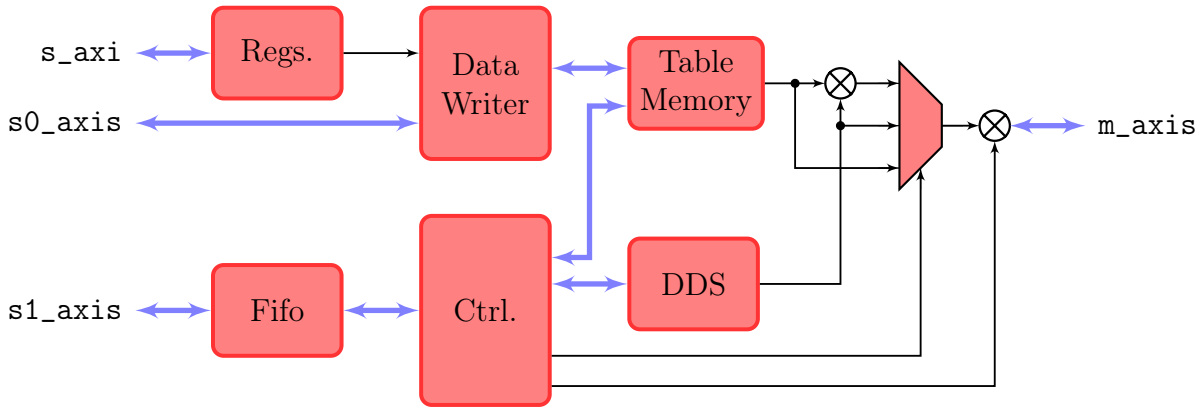

Figure 2: Signal Generator block diagram.

the fields of the input `s1_axis` interface is as follows:

79	78	77 : 76	75 : 64	63 : 48	47 : 32	31 : 16	15 : 0
`stdysel`	`mode`	`outsel`	`nsamp`	`gain`	`addr`	`phase`	`freq`

There are some fields that are used for the DDS section, other fields for the Table section, and other are shared. The `freq` field is 16-bit and should be used to specify the frequency of the DDS block. DDS frequency is $[0-500]$ MHz as the block runs at that speed and it is a complex signal generator. The configuration of the frequency is an unsigned integer number that goes from 0 to 65535, and maps into the frequency range. The `phase` field is also a 16-bit integer number, and can be used to specify the starting phase of the DDS signal. It's important to clarify how the Signal Generator V2 handles the phase. This block keeps track of the phase to implemente the so called phase coherency. Let's say a signal of frequency ω_0 is selected. This signal could be written mathematically as:

$$x_0(t) = a_0 \cos(\omega_0 t + \theta_0).$$

Now let's say another signal of frequency ω_1 is needed, starting at time t_1. What is the phase relationship between $x_0(t)$ and $x_1(t)$? The phase-coherent signal generator will compute the initial phase of the signal $x_1(t)$ to be $\omega_1 t_1 + \theta_1$. Simply put, the generator works as if any selected frequency is referenced to the same initial phase. If a signal of a given frequency is used for a certain time, then a different frequency, and back to the

same frequency with the same initial phase, both signals will meet as if they were coming from an always-running generator.

Let's continue with the other parameters. The `addr` field refers to the first sample of the table section. This memory should be loaded with the desired waveform before usage. The address is incremented automatically from the start address. The `gain` field is a signed, 16-bit number which applies a gain to the overall output. This number goes from -32768 to 32767, and corresponds to ± 1. The next field `nsamp` is used to specify the length or number of samples of the waveform. It is 12-bit and allows to specify waveforms up to 4096 samples long. The field `outsel` is used to select the output source:

- `outsel=0`: output is the product of table and DDS.
- `outsel=1`: output is coming from the DDS section.
- `outsel=2`: output comes from the table for real part, 0 for imaginary part.
- `outsel=3`: output is set to 0.

The `mode` is a 1-bit field and allows to specify "periodic" or "one-shot" mode. When this bit is set to 0, the signal generator will create a waveform with the specified number of samples and then will either read the next waveform or end if no more waveforms are in the queue. If this mode is set to 1, the signal generator will keep repeating the actual waveform if the queue is empty, or read the next waveform if the queue is not empty, after the number of samples have been completed. As an example, let's say a pure sine wave wants to be created with "infinite" duration. The user could push a configuration using DDS for output selection and the required number of samples, say 100 in this case. If the mode is set to 1, the signal generator will keep repeating the same waveform, letting the DDS section to run freely without any phase jumps. Then, later on, the user could push another waveform into the signal generator with the mode bit set to 0. After completing the actual period of 100 samples of the infinite wave, the signal generator will read from the queue the new waveform and output it. Once completed, the block will stop until new waveforms are pushed into it. There is an extra bit added to V2 of the signal generator. The field `stdysel` allows to select weather the last value after completion of all waveforms

is the last sample or a zero-valued number. When this bit is set to 0, the last sample is kept in the output of the signal generator. When this bit is set to 1, the signal generator will output a zero-valued sample once all waveforms have been completed.

The actual implementation allows to queue up to 16 waveforms. The provided axi-stream interface will signal "not ready" when the block is either full or not ready to receive new waveforms. Note that when using this signal generator with the tProcessor block, the user should be careful not to push too many waveforms and cause the signal generator to be full. This situation may lead to lost waveforms as the actual implementation of the tProcessor does not wait until the slave is ready to complete the transaction.

5 Code examples

This section provides some code examples to introduce the usage of the tProcessor language. Examples are introductory and may not cover all available options.

Basic register operations

Registers are used in almost all instructions. The tProc provides 4 pages, with 16 registers each page. Register number 0 is always zero, so there are 15 effective registers for a total of 60 registers available for usage. Let's start with a simple register set example:

```
1  // Initialize registers.
2  regwi 0, $1, 1234; // freq = 1234
3  regwi 0, $2,  100; // delta freq = 100
4  regwi 0, $3,   10; // loop register
5
6  // Loop.
7  LOOP: math   0, $1, $1 + $2; // freq = freq + delta freq
8        loopnz 0, $3, @LOOP;
9
10 // Program end.
11 end;
```

Let's walk through this example to understand how the language work. Lines 2, 3 and 4 load initial values on registers. In this case register 1 on page 0 is loaded with the value 1234, register 2 with the value 100 and register 3 with the value 10. Line 7 shows how to label instructions for jump operations. In this case the program memory address of the instruction on line 7 will be the destination address for the `loopnz` instruction. Line 7 makes use of the `R-type` instruction `math`, which uses 2 source registers and 1 destination register. In this case, the values of registers 1 and 2 is added and written back into register 1. This simple example shows how to implement a delta variation on a variable. Line 8 will use register 3 on page 0 as the loop register. The instruction will jump to address on line 7 if the loop register is different from zero, and decrement the value of the register before jumping. It is important to note that instruction on line 7 will be executed 11 times and not 10. Line 11 finishes the execution of the program.

Nested loops and stack usage

Let's move to a more complex example, using the stack for nesting loops. Imagine an algorithm is needed to sweep three variables: frequency, gain and number of samples. This variables will have an initial value and associated increment. The following listing shows a possible implementation using the stack to nest the three loops:

```
1  // Page 0:
2  // $1 : df
3  // $2 : freq
4  // $3 : dg
5  // $4 : gain
6  // $5 : dn
7  // $6 : nsamp
8  // $7 : loop register
9
10 // i = 5 (outer-most loop)
11 // j = 3
12 // k = 6 (inner-most loop)
```

```
13
14 // Initialize deltas.
15 regwi 0, $1,    -50; // delta freq = -50
16 regwi 0, $3,     34; // delta gain = 34
17 regwi 0, $5,    100; // delta nsamples = 100
18
19 regwi 0, $2, 12000; // freq = 12000
20 regwi 0, $7,      5; // i = 5
21
22 // Loop freq.
23 FREQ:  math  0, $2, $2 + $1; // freq = freq + df
24        regwi 0, $4,      120; // gain = 120
25        pushi 0, $7, $7,    3; // i -> stack, j = 3
26        // Loop gain.
27        GAIN: math  0, $4, $4 + $3; // gain = gain + dg
28              regwi 0, $6,       25; // nsamples = 25
29              pushi 0, $7, $7,    6; // j -> stack, k = 6
30              // Loop nsamp.
31              NSAMP: math   0, $6, $6 + $5; // nsamp = nsamp + dn
32                     loopnz 0, $7,  @NSAMP; // loop k
33              popi   0, $7;          // stack -> j
34              loopnz 0, $7,  @GAIN; // loop j
35        popi   0, $7;          // stack -> i
36        loopnz 0, $7,  @FREQ; // loop i
37
38 // Program end.
39 end;
```

The example shows three loops, named i, j and k. These three loops are implemented using a single register, which is register number 7 on page 0. The stack is used to push/pop and allows to nest the loop very easily. Lines 15, 16 and 17 initialize the delta

registers, and lines 19 and 20 initializes the registers used for frequency and outer loop. Indentations are added to ease the reading of the algorithm and they do not have any effect in the generated code.

The frequency loop starts with the addition of registers number 2 and 1 on page 0, writing the result back to register 2. As in the previous example, this allows to keep incrementing the value of the variable re-using the same register for source and destination. This line is a labeled instruction as will be later used for jump. Line 24 initializes the gain register, and the next line pushes the contents of register 7 to the stack, and moves the value 3 into the same register. Note how a single instruction allows to push the register to the stack and replace its value using the immediate field. What follows is basically the same structure for the inner loops. Line 27 computes delta gain, and lines 28 and 29 load the number of samples and push the contents of the loop register to the stack to initialize the inner-most loop value to 6. Note that after instruction 29 is executed, the stack has two values: i and j. The latter is in the top of the stack, and the former at the bottom. Lines 31 and 32 implement the loop over number of samples variable. Once this loop has been completed, instruction on line 33 is executed, which pops the value of j from the stack into register 7. Instruction on line 34 will jump back to `GAIN` label if the value is different from zero, decrementing its value previously. Once the second loop has been completed, lines 35 and 36 repeat the same structure to recover the outer-most loop counter value and repeat accordingly. This simple example shows how to implement a 3 variables increment algorithm, using a single register for loop counter and the stack to nest the loops.

Timed instructions and sychronization

The previous examples did not make use of the timed-instruction mechanism. The following listing shows a simple example of use of timed-instructions and synchonization mechanism:

```
1 // Initialize registers.
2 regwi 0, $1,  78; // out = 78
```

```
3  regwi 0, $2, 200; // i = 200 (loop)
4  regwi 0, $3, 333; // T = 333
5
6  // Loop.
7  LOOP: seti   0,  0, $1, 20; // ch 0 = out @t = 20
8        mathi  0, $1, $1 + 1; // out = out + 1
9        synci              50; // toff = toff + 50
10       loopnz 0, $2,  @LOOP; // loop i
11
12 // Finish.
13 sync 0,           $3; // toff = toff + $3
14 math 0, $1, $0 + $0; // out = 0
15 seti 0,  0, $1,  55; // ch 0 = out @t = 55
16
17 // Program end.
18 end;
```

Lines 2, 3 and 4 have already been explained on great detail and they initialize register values. Line 7 shows the first timed-instruction. This instruction will set the contents of register 1 on page 0 to be the output on channel 0, at time 20. What is the meaning of time here? When the processor starts executing the code, the internal master clock counter starts counting and never stops. The time offset register is reset to zero at this time. Instruction on Line 7 will be scheduled to happen at time 20 the first time, which means when the master clock reaches that value the output port will be written. Line 9 uses the `synci` instruction to synchronize the internal time offset to 50 units. The second pass of the loop will decode instruction on Line 7 again, but the absolute time at which this instruction is scheduled equals $50 + 20 = 70$ units. However, with respect to the new reference that was set at 50 units by Line 9, the instruction happens 20 time units after as in the first case. This allows to think the algorithm in terms of small timed pieces by inserting time references using synchronization instructions. After the second pass, the instruction `synci 50` will set the internal time offset to its previous value plus 50, which

results on $t_{off} = 100$ time units. Instruction on Line 7 will now be scheduled to happen at 120. The process is repeated until the loop is completed.

Once the loop finished, Line 13 uses the `R-type` flavor of the synchronization mechanism. The end result is the same, but instead of using a immediate value, source register 3 on page 0 is used in this case. Line 15 sets the output again, which will occur 55 time units after the previous synchronization. Even if the absolute time of the instruction is difficult to say without going on a bit of math, what matters is that the time structure is very simple: 200 repetitions of instructions happenning 20 time units after the reference for the loop, and one more instruction at time $T + 333 + 55$. The value of T is simply the product of the number of times the loop is repeated by the synchronization time.

Math, bit-wise and memory access

Mathematical and bit-wise operations were added in the 64-bit versino of the tProcessor. Data memory was also added. In the following pages, few examples of use of these new kind of instructions will be shown.

```
1  // Initialize registers.
2  regwi 0, $1,  100; // freq = 100
3  regwi 0, $2,   15; // df = 15
4  regwi 0, $3, 1000; // loop
5
6  // Loop.
7        regwi 0, $4,       0; // i = 0
8  LOOP: math  0, $5, $4 * $2; // temp = i*df
9        math  0, $6, $1 + $5; // freq = freq + i*df
10       mathi 0, $4, $4 +  1; // i = i + 1
11       condj 0, $4 < $3, @LOOP;
12
13 // Program end.
14 end;
```

This simple program shows how to use product to implement a delta variation over a variable with a loop counter. Lines 2, 3 and 4 initialize frequency, delta frequency and the maximum number of repetitions. Line 7 initializes the register 4 on page 0 to be 0, as it will be used for counting and loop variable. Line 8 computes the accumulated delta frequency as $i * \Delta f$. The first time register 4 is initialized to 0, and this product will lead to 0. The next instruction adds the value of the initial frequency, held by register 1, with the accumulated delta frequency. Line number 10 increments the value of the loop register by 1, and finally line number 11 implements a loop using a conditional instead of a loop instruction. This line will jump to line 8 of code while register number 4 is smaller than register number 3, which was initialized to the maximum count value.

An interesting example is the interface with the Signal Generator V2 block, as it uses 5 registers for programming the block and 1 extra register for time specification of the set instruction. Also, bit-wise operations are needed to ease creating the value of the mode bits of the signal generator:

```
1  // Signal Generator V2.
2  // 15..0  : frequency.
3  // 31..16 : phase.
4  // 47..32 : addr.
5  // 63..48 : gain.
6  // 75..64 : nsamp.
7  // 77..76 : outsel (00: product, 01: dds, 10: table, 11: zero-value).
8  //     78 : mode   (0: nsamp, 1: periodic).
9  //     79 : stdysel (0: last value, 1: zero-value).
10
11         // Registers.
12         regwi 0, $1,      750; // freq.
13         regwi 0, $2,        0; // phase.
14         regwi 0, $3,        0; // addr.
15         regwi 0, $4,    10000; // gain.
16         regwi 0, $5,     1000; // nsamp.
```

```
17          regwi 0, $6,      0x8; // 0b1000.
18          bitwi 0, $6, $6 << 12; // left shift register 6 by 12 bits.
19          bitw  0, $5,  $5 | $6; // r5 = r5 or r6.
20          regwi 0, $7,       20; // t = 20.
21
22          // Signal start.
23          regwi 1, $1,    0x1;
24          seti  0, 1, $1, 200;
25          synci           200;
26
27          // Generate waveforms.
28          regwi  1,  $2, 13;                      // loop counter.
29          regwi  0, $10,  0;                      // memory address index.
30  LOOP0:  set    1,   0, $1, $2, $3, $4, $5, $7; // out @t = $7.
31          mathi  0,  $4, $4 + 300;                // gain = gain + 300.
32          memw   0,  $4, $10;                     // mem[$10] = $4
33          mathi  0, $10, $10 + 1;                 // memory address +1
34          loopnz 1,  $2, @LOOP0;
35
36          // Signal end.
37          regwi 1, $1,    0x0;
38          seti  0, 1, $1, 500;
39
40          // Program end.
41          end;
```

This listing is a little bit more complicated than the previous ones and uses most of the available instructions. Lines 12 through 15 are used to initialize frequency, phase, address and gain registers. These are going to be used to program the Signal Generator V2. Line 16 sets the number of samples, which are the lower 12 bits according to the specification of the signal generator. Line 17 sets the 4 mode bits of the signal generator. These bits

should be placed into the 4 most-significant bits of the configuration word. To achieve that, line 18 moves these 4 configuration bits to the left using the left shift bit-wise operation. Oncde the bits are placed in the right spot, line 19 performs the bit-wise or operation to combine the number of samples and the extre 4 configuration bits. Register number 7 on line 20 will be used to set the time of the instruction.

Lines 23, 24 and 25 are used to set the least-significant bit of channel 0 at time 200. This is simply to create an external reference to trigger equipement. Line 28 initializes the loop counter to 13. Line 34 uses this register to implement the loop. Line 29 initializes an extra register that will be used to address the data memory. Line 30 is the set instruction, which allows using 5 registers for output data plus a 6th register to specify time. In this case, the signal generator is connected to channel 1 and this is why the set instruction is using this channel. Line 31 increments the gain register and line 32 writes this gain value into the data memory, using register number 10 for the destination address, which is incremented in the following line.

There is an interesting detail on this loop. As it can be seen, there is no synchronization instruction inside the loop, which means that all timed-instructions will be scheduled to happen at the same time. This is not a problem due to the fact that the signal generator can queue up to 16 waveforms. The result of this loop will be pulses of varying gain one after each other, without any gap in between. This trick can be used to output pulses wihout any delay between them. The option using a synchronization instruction could be used when a gap between the waveforms is needed.