# Sequencer Documentation

## Contents

# 1. Introduction

A sequencer is a microprocessor implemented on an FPGA that is used for controlling experimental setups. Using a single cycle microprocessor design, it features the exact clock timing in the resolution of 10ns. Also a sequencer can run any program that is written with the set of instructions, making it applicable to different experiments. It also has sufficient output ports that are capable of sending signals to external devices, which allows the central controllability for different experimental setups. The software layer on a host PC used for communication between the PC and a sequencer uses Python language, taking the advantage of simple programming.

This documentation is a tutorial on how to use a sequencer. It is aimed for those who have no experience on using a sequencer. This document deals the overall design of the sequencer, basic setup processes, and details of sequencer programming, and should be sufficient enough to make experimental setups from an empty Arty S7 board. Details required for developing purposes are presented on the Appendix. It is intended to be as self-contained as possible, but basic understanding of Python programming and microprocessor architecture is required. Also, understanding of assembly languages, experiences on FPGA boards and experiments may be helpful, but not crucial.

The version of the sequencer described in this document is v4_01.

# 2. Overview

### 1) Overall Sequencer System

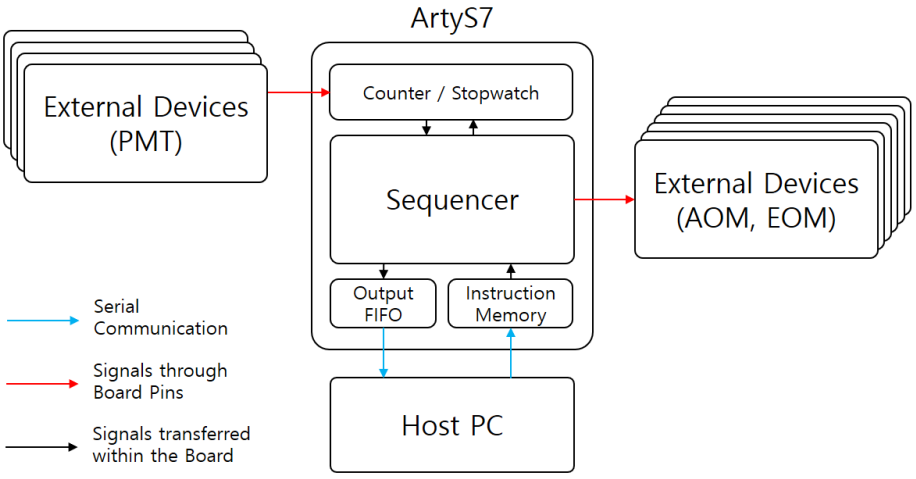


**Figure 1 Sequencer System**

Figure 1 shows the overall sequencer system. A sequencer and peripheral modules (counters and stopwatches) are implemented on an Arty S7 FPGA board. The board communicates with the host PC via serial communication. The PC sends instructions to the board and the board sends data to the PC. The sequencer is connected to external devices through output pins. There are also inputs which come from outside of the board, which are used for stop signals of stopwatches or inputs of counters.

### 2) Sequencer Design

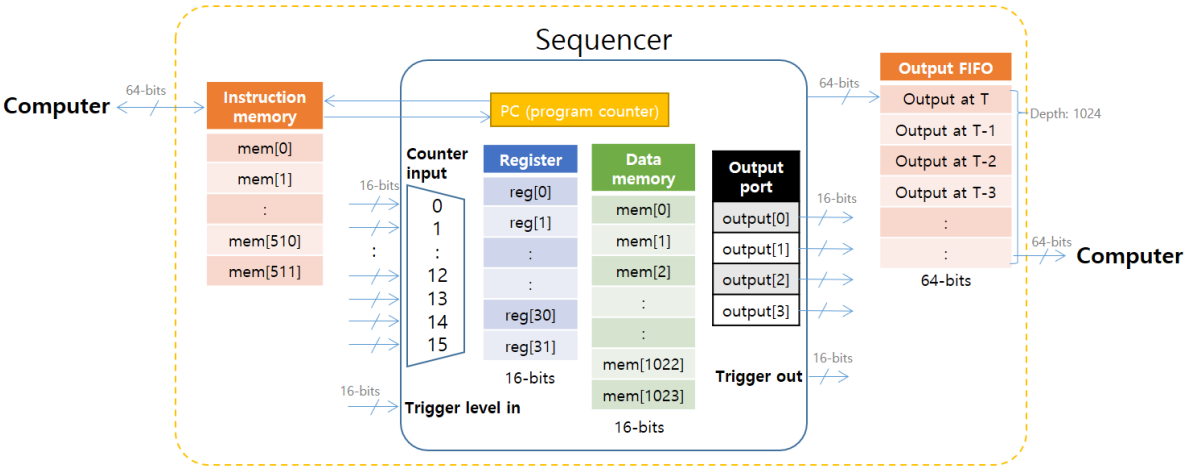


**Figure 2 Overall Design of Sequencer**

Focusing on the sequencer module itself, Figure 2 shows the overall design of the sequencer module. There are several submodules that constitute a sequencer module. Instruction memory, PC, register, and data memory have the similar role with other microprocessors. A sequencer runs on an 100MHz clock, or in other words, one clock cycle is 10ns. It is a single cycle processor of which the execution of instructions (except wait instructions) are done in one cycle. Output FIFO is used as a buffer of data that is sent to the host PC.

There are two types of outputs in a sequencer module. One is a normal output composed of four ports, usually denoted as `output[0]~[3]`. Each port outputs a 16-bit signal. The other is a single 16-bit trigger output which generates only one clock pulses. The former is used to send signals outside the board or to enable counters on the board. The latter is used for reset signals of counters or start/reset signals of stopwatches.

There are also two different types of input. One is a so-called trigger-level-in input. It is a single 16-bit input which is used for aborting conditional waits (see Example 8 for details). The other type of input is a counter input. There are 16 counter ports in total, and each port is a 16-bit length input.

In fact, not all 16 counter ports are used for input ports. That is, `counter[14]` and `counter[15]` have predefined roles. `counter[14]` is used for reading the value of trigger-level-in, and `counter[15]` is used for reading remaining number of cycles when a wait instruction was aborted. The other remaining ports, `counter[13:0]`, receives input from outside the module.

The below is the short summary of the specification of the sequencer.

**Memory**

- Instruction memory: 512 x 64-bits
- Registers: 32 x 16-bits
- Data memory: 1024 x 16-bits

**Data transfer to computer**

- Output FIFO: (1024 depth) x (64-bits)

**Inputs**

- Counter input: (14 ports) x (16-bits)
  * `counter[14]` and `counter[15]` have predefined roles.
- Trigger-level-in input: 16-bits

**Outputs**

- Output port: (4 ports) x (16-bits)
- Trigger output: 16-bits

**Speed**

- Clock speed: 100 MHz
- Single-cycle processor: execution of each instruction takes only 1-clock except wait instruction

#### 3) Software

The software part of sequencer which interprets instructions and send them to the board is implemented and run on the host PC. The user can write a sequencer program on the host PC and then the PC will load the program on the sequencer. After the sequencer execution, the user can fetch the data sent back from the sequencer and post-process them for analysis. Detailed procedure of sequencer programming is described throughout this document.

## 3. Getting Started

First step to use a sequencer is to load the bitstream on an Arty S7 FPGA board. The bit file used

for sequencer is ₩₩172.22.22.101₩qc_user₩Users₩thkim₩Arty_S7₩Sequencer₩v4_01₩v4_01.runs₩impl_1₩main.bit. You can use Vivado to load the file on the board. Also if you store the file in the flash memory, you can reload the bitstream without having to reprogram.

Once the bitstream is loaded on the board, you can run the sequencer program. First, open the `FirstSequencerExample.py` and find the line 25 '`sequencer = ArtyS7('COM4')`'. Here, you need to edit the serial port `'COM4'` with the actual port number that connects your PC and the Arty S7 board. Finally run `FirstSequencerExample.py` on Python. If done correctly, you will have `[0, 0, 0, 10]` printed on your PC.

## 4. Basics of Sequencer Programming

In this section, the basics of sequencer programming will be explained. Specifically, the overall structure of the python sequencer program will be explained using the example code of the previous section.

```
#  1  # Import required modules for sequencer programming
#  2  from SequencerProgram_v1_07 import SequencerProgram, reg
#  3  import SequencerUtility_v1_01 as su
#  4  from ArtyS7_v1_02 import ArtyS7
#  5  import HardwareDefinition_type_EX as hd
#  6
#  7  # Assembly Codes
#  8  s = SequencerProgram()
#  9
# 10  s.load_immediate(reg[0], 0)
# 11  s.write_to_fifo(reg[0], reg[0], reg[0], 10, 'First Sequencer Code')
# 12  s.stop()
# 13
# 14  if __name__ == '__main__':
# 15      # Send Instructions and Start Sequencer
# 16      if 'sequencer' in vars():
# 17          sequencer.close()
# 18      sequencer = ArtyS7('COM4')
# 19      sequencer.check_version(hd.HW_VERSION)
# 20
# 21      s.program(show=False, target=sequencer)
# 22
# 23      sequencer.auto_mode()
# 24      sequencer.start_sequencer()
# 25
# 26      # Data Acquisition
# 27      data = []
# 28
# 29      while(sequencer.sequencer_running_status() == 'running'):
# 30          data_count = sequencer.fifo_data_length()
# 31          data += sequencer.read_fifo_data(data_count)
# 32
# 33      data_count = sequencer.fifo_data_length()
```

```
# 34      while (data_count > 0):
# 35          data += sequencer.read_fifo_data(data_count)
# 36          data_count = sequencer.fifo_data_length()
# 37
# 38      for each in data:
# 39          print(each)
```

**FirstSequencerExample.py**

A sequencer program consists of the following four parts.

### 1) Import (Line 1~5)

In the 'Import' part, required Python modules must be imported. Generally

```
from SequencerProgram_v1_07 import SequencerProgram, reg
import SequencerUtility_v1_01 as su
from ArtyS7_v1_02 import ArtyS7
import HardwareDefinition_type_EX as hd
```

would be sufficient. And for the import to be done properly, the corresponding python modules must be present on the same directory.

### 2) Assembly Code (Line 7~12)

The assembly code part specifies which instructions the sequencer should execute. The instructions required for sequencer operations are defined as methods of the `SequencerProgram` class. By writing assembly codes using methods of the `SequencerProgram` class, the instructions are saved as a list in the instance. Note that until this step, the instructions are only stored in the host PC. Converting the instructions into machine codes and sending them to the sequencer are done in later steps.

Here, and in later examples, the instructions that are presented newly will be explained. For the list of full instructions, see 'Summary of Instructions'.

`s.load_immediate()` method saves the immediate value in the destination register. That is, `s.load_immediate(reg[0], 0)` in this example makes the value of register 0 into 0. This instruction is frequently used to initialize registers.

`s.write_to_fifo()` method is used to send data from the sequencer to the host PC through the output FIFO. One FIFO entry consists of four data, which are three register values and one event label integer. All values are in 16 bits, and therefore one entry of FIFO has 64-bit length. An event label can be an arbitrary 16-bit integer. This can be used to distinguish data stored for different reasons. The data stored in FIFO will be fetched by the host PC in the 'Data Acquisition' step. In this example, `reg[0]` has the value of 0, and therefore `[0, 0, 0, 10]` will be written into the FIFO.

The fifth argument `'First Sequencer Code'` is a comment that helps understanding the code. Every instruction method can have additional one argument as a comment.

`s.stop()` method is, as the name indicates, the method that stops the execution of the sequencer. It is in fact mandatory to have this instruction at the end of the assembly code, because otherwise the sequencer will not terminate.

### 3) Send Instructions and Start Sequencer (Line 15~24)

Once the assembly programming is finished, the instructions should be sent to the sequencer and executed. First, an `ArtyS7` instance is made with the serial port specified (Line 18). `sequencer.check_version(hd.HW_VERSION)` method compares the version of the sequencer loaded on the FPGA and the version of hardware description used for programming. If the versions do not match, it raises an exception.

The main action of `s.program()` method is decoding the assembly codes into machine codes and sending the decoded codes to the sequencer. In order to load programs on the board, you have to set the `target` argument as `sequencer` (an `ArtyS7` instance with specified serial port). There are several other arguments of s.program() method. If you want to see the program in human-readable outputs, set the `show` argument to `True` (which is the default). Additionally, setting the `machine_code` argument to `True` shows the machine code of the program. You can also save the hex file if you specify the file directory on the `hex_file` argument.

When using a sequencer, there are two modes that can change the port output of Arty S7. One is the manual mode and the other is the auto mode. In the manual mode, the change of the output is done by sending protocol commands to the Arty S7 board. In the auto mode, the output changes by the instructions of the sequencer. Since the latter mode is the wanted action, you have to call `sequencer.auto_mode()` in order to set the sequencer into the auto mode (the default mode of the sequencer is the manual mode).

Once the program is loaded onto the board and the board is set to the auto mode, you can start the sequencer execution by calling `sequencer.start_sequencer()`.

### 4) Data Acquisition (FIFO communication) (Line 26~36)

Once the sequencer starts running, you need to fetch the data sent from the sequencer. `sequencer.read_fifo_data(data_count)` method outputs the data stored in FIFO. Note that you also need to fetch the data when the sequencer is running in order to prevent FIFO overflow

(Line 29~31). After the sequencer finishes running, you fetch the leftover data (Line 33~36). Then you can post-process (e.g. print, plot) the data stored.

In this example, the result should be `[0, 0, 0, 10]`, the result stored in Line 11.

## 5. Basic Operations

In this section, basic assembly operations are introduced with examples. To save the space, only the assembly code part will be presented. (All codes are in `SequencerExample.py` file. You can comment/uncomment necessary parts to follow these examples. Also don't forget to edit the serial port number in this file too.)

### 1) Example 1: Arithmetic Operations (add, addi, sub, subi)

```
#  1  # Example 1: Arithmetic Operations
#  2  s = SequencerProgram()
#  3
#  4  s.load_immediate(reg[0], 0)
#  5  s.add(reg[0], reg[0], 1, 'reg[0] <= reg[0] + 1')
#  6  s.add(reg[1], reg[0], reg[0], 'reg[1] <= reg[0] + reg[0]')
#  7  s.subtract(reg[2], reg[1], reg[0], 'reg[2] <= reg[1] - reg[0]')
#  8  s.subtract(reg[2], reg[2], 2, 'reg[2] <= reg[2] - 2')
#  9  s.write_to_fifo(reg[0], reg[1], reg[2], 10, 'Arithmetic Example')
# 10  s.stop()
```

There are four arithmetic operations for the sequencer, add, addi, sub, and subi. However, using the advantage of Python language, there is no separate method for add(sub) and addi(subi). By differing the datatype of third argument, the method automatically differentiates the two instructions. That is, you can use either `s.add(reg[n], reg[m], reg[l])` or `s.add(reg[n], reg[m], value)` form when you do the sequencer programming. `s.subtract()` method has the similar form.

In Line 4, `reg[0]` stores 0. In Line 5, 1 is added to `reg[0]`, which as a result becomes 1. In Line 6, `reg[1]` stores 2 (=`reg[0]+reg[0]=1+1`). In Line 7, `reg[2]` stores 1 (=`reg[1]-reg[0]=2-1`). In Line 8, `reg[2]` should store `reg[2]-2=-1`, but the sequencer uses 16-bit unsigned interger representation, and therefore `reg[2]` stores 65535(=$2^{16}-1$) instead. The final result fetched from the output FIFO is `[1, 2, 65535, 10]`.

### 2) Example 2: Branch/Jump Operations (blt, blti, beq, beqi, jump)

```
#  1  # Example 2: Branch/Jump Operations
#  2  s = SequencerProgram()
#  3
```

```
#  4  s.load_immediate(reg[0], 0)
#  5  s.load_immediate(reg[1], 1)
#  6
#  7  s.add_value = \
#  8  \
#  9  s.add(reg[0], reg[0], reg[1], 'reg[0] += reg[1]')
# 10  s.add(reg[1], reg[1], 1, 'reg[1]++')
# 11  s.branch_if_less_than('add_value', reg[1], 11)
# 12
# 13  s.write_to_fifo(reg[0], reg[0], reg[1], 10, 'Branch/Jump Example')
# 14  s.stop()
```

There are four branch (conditional jump) operations and one (unconditional) jump operation for the sequencer. Branch operations contain blt(branch-if-less-than), blti(branch-if-less-than-immediate), beq(branch-if-equal), beqi(branch-if-equal-immediate). These operations can be used for loops or conditional operations.

`s.branch_if_less_than()` method is the method representation of blt and blti instructions. Similar to arithmetic operations, the two instructions are differentiated by the data type of the third argument. `s.branch_if_less_than()` method compares the second and third arguments and if the second argument is smaller than the third, the sequencer branches to (that is, the next instruction becomes) the instruction of which the address is specified by the first argument. Otherwise, the sequencer proceeds on executing following instructions. `s.branch_if_equal()` is the method for beq and beqi instructions. It branches when the second and third arguments are equal. `s.jump()` method takes only one argument for instruction address, in that it is an unconditional branch(jump).

Line 7~11 in the above example shows a loop in which `reg[0]` adds each value stored in `reg[1]` while `reg[1]` increases from 1 to 10. Note that once `reg[1]` becomes 11, it does not branch back and therefore 11 is not added to `reg[0]`. As a result, the data sent from FIFO is `[55, 55, 11, 10]`.

The point that needs to be taken care is the syntax of branches. In the example, you can see that the first argument of `s.branch_if_less_than()` method is a string('`add_value`'), and this indicates that it must branch to `s.add_value` of Line 7. Every instruction method of `SequencerProgram` returns the instruction address of itself. Therefore, the address of `s.add()` in Line 9 is stored in the attribute `s.add_value`. During the programming procedure, string addresses are converted to the instance attributes of which the names are the string addresses. In fact, you can also provide the target address in terms of address numbers or class attributes, but it is highly recommended that you use the string name for ease of programmability and avoidance of other potential errors. (Note that the address of Line 9 is 2, not 9, because it is the third instruction (indexed from 0).) Other branches and jump methods are used similarly.

### 3) Example 3: Memory Access (load_word, store_word)

```
#  1  # Example 3: Memory Access(Load, Store)
#  2  s = SequencerProgram()
#  3
#  4  s.load_immediate(reg[0], 0)
#  5  s.load_immediate(reg[1], 0)
#  6
#  7  s.repeat_store = \
#  8  \
#  9  s.add(reg[0], reg[0], 2)
# 10  s.store_word_to_memory(reg[1], reg[0])
# 11  s.add(reg[1], reg[1], 1)
# 12  s.branch_if_less_than('repeat_store', reg[1], 3)
# 13
# 14  s.load_immediate(reg[0], 0)
# 15  s.load_word_from_memory(reg[1], reg[0])
# 16  s.add(reg[0], reg[0], 1)
# 17  s.load_word_from_memory(reg[2], reg[0])
# 18  s.add(reg[0], reg[0], 1)
# 19  s.load_word_from_memory(reg[3], reg[0])
# 20
# 21  s.write_to_fifo(reg[1], reg[2], reg[3], 10, 'Store/Load Example')
# 22  s.stop()
```

There are two instructions that access the data memory of the sequencer, load_word and store_word. load_word instruction loads the values stored in the memory into the target register. store_word instruction saves the value of a register in the memory.

Line 7~12 shows the usage of `s.store_word_to_memory()` method. The method has two arguments. The first argument is the memory address register and the second argument is the source register. The value of the source register is stored in the memory of which the address is the value of the memory address register. When the store instruction in Line 10 is first executed, `reg[1]` has the value 0 and `reg[0]` has the value 2. Since `reg[1]` is the memory address register and `reg[0]` is the source register, 2 is saved in `mem[0]`[1]. Since Line 7~12 are executed three times with `reg[0]` and `reg[1]` incremented by 2 and 1 respectively in each iteration, 4 is stored in `mem[1]` (second iteration) and 6 is stored in `mem[2]` (third iteration).

`s.load_word_from_memory()` method has two arguments. First argument is the target register and the second argument is the memory address register. The value stored in memory, of which the address is the value of the memory address register, is loaded into the target register. Since the memory address register `reg[0]` has the value 0 in Line 15, the sequencer fetches the value stored in `mem[0]` and loads it in `reg[1]`, the target register. Line 17 and Line 19 are executed similarly.

[1] Throughout this document, `mem[n]` denotes the memory with the address n.

That is, the value stored in `mem[1](mem[2])` is loaded in `reg[2](reg[3])`. Recalling the values stored in memory in Line 7~12, `reg[1], reg[2], reg[3]` will have 2, 4, 6 respectively. The data sent through the FIFO agrees with this result (`[2, 4, 6, 10]`).

# 6. Extending Sequencer

So far, only the Arty S7 board alone was used. However, the main usage of the sequencer comes from utilizing peripherals and controlling external devices. In this section, how external devices are connected to and controlled by the sequencer is explained. To explain with tangible examples, a very simple circuit with two LEDs and two push buttons will be used.

### 1) Pmod Connections and Hardware Definition

An Arty S7 board uses Pmod connectors to send and receive signals from outside. The Pmod Connectors are the part marked 15 in Figure 3, and these connectors use 3.3V power source. Among the four Pmods, the sequencer uses two ports, namely ja and jb, the left two ports in the figure. Excluding Vcc and GND, each connector has 8 pins, which are labeled from ja(jb)_0 to ja(jb)_7 as shown in Figure 4. Among the 16 available pins of ja and jb connectors, jb_0, jb_2, jb_4, jb_6, ja_2 are used for input pins and the other pins are all used for output pins.

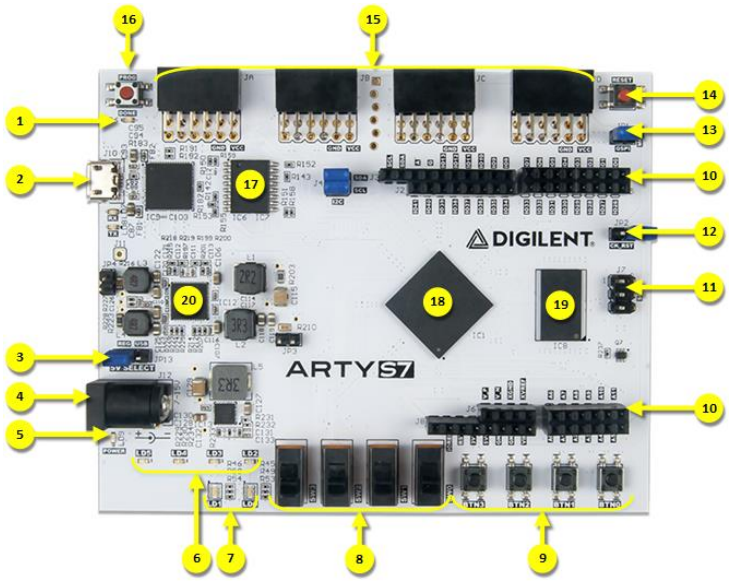


**Figure 3 Arty S7 Features[2]**

[2] Photo Reference: Arty S7 Reference Manual
(https://reference.digilentinc.com/reference/programmable-logic/arty-s7/reference-manual)

| | | | | | |
|---|---|---|---|---|---|
| Vcc | GND | ja_3 | ja_2 | ja_1 | ja_0 |
| Vcc | GND | ja_7 | ja_6 | ja_5 | ja_4 |

**Figure 4 Pin Labels for Pmod ja (Same for jb)**

To be used in later examples, the circuit of Figure 5 will be used. That is, 2 LEDs are connected to jb_1 and jb_3 pins, and 2 push buttons are connected to jb_0 and jb_2 pins.

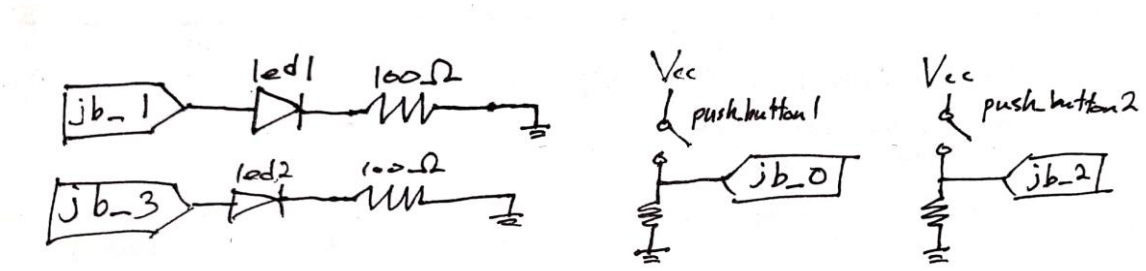


**Figure 5 Example Circuit**

Once you finish the wiring, you have to make a `HardwareDefinition_type_*.py` file that contains the mapping information. In this example, `HardwareDefinition_type_EX.py` file will be used.

```
# 10  from HardwareDefinition_v4_01 import *
# 11
# 12  # Input port pin mapping
# 13  # For input pin, there will be only one driver
# 14  input_mapping = {'jb_0': 'push_button1', 'jb_2': 'push_button2'}
# 15
# 16  # Output port pin mapping
# 17  # For output pin, there might be more than one device controlled by
      the output pin
# 18  output_mapping = {'LED1' : 'jb_1', 'LED2' : 'jb_3'}
```

**HardwareDefinition_type_EX.py Line 10~18**

The part you have to focus in `HardwareDefinition_type_EX.py` is Line 10~18. In Line 10, it imports a `HardwareDefinition_v4_01` module. The version of the module you import must match the version of the sequencer that is loaded on the FPGA board.

Line 14 defines the input mapping of the sequencer. It is stored as a dictionary data type with the pin names being keys and connected hardware being values. Similarly, Line 18 defines the output mapping of the sequencer, but in this case, connected hardware are used as keys and the corresponding pins are used as values.

The rest of the `HardwareDefinition_type_EX.py` is adding aliases to pin variables according to the input and output mapping. For example, `push_button1_counter_reset` is added as an

alias of `jb_0_counter_reset` and `LED1_out` is added as an alias of `jb_1_out`. The usage of each pin variable will be shown in later examples.

### 2) Outside the Sequencer

A sequencer module has four output ports, each having 16-bit output. Among the four ports, only the first two ports have predefined roles. The first output port `output[0]` is an external output port. It is connected to external devices (e.g. AOM, EOM) and is generally used to activate/deactivate the devices. The second port, `output[1]`, is a counter control port. A counter module, which will be explained very soon, requires an enable signal to be activated. Each bit of the counter control port is connected to the enable signal of different counter modules, and setting each bit as 1(0) will start(stop) counting input signals.

There are two types of peripherals that extends sequencer design, counters and stopwatches. The block diagram of each module is shown in Figure 6. A counter is a module that counts the number of input signals (or posedges of a clock) when the enable signal(`clkEnable`) is high. There is a reset signal which sets the count back to zero. A stopwatch outputs the number of clocks past after it received a start signal. The role of a stop/reset signal is evident from its name. A stopwatch module has another output called stopped, which indicates whether the stopwatch has been stopped by a stop signal. One thing to note about the stopwatch is that while the sequencer runs on 100MHz clock, stopwatches run on 800MHz clock. That is, one clock cycle of the sequencer is equivalent to eight cycles of a stopwatch.

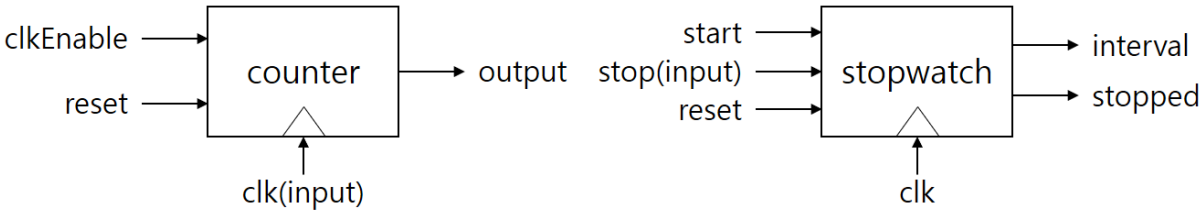


**Figure 6 Block Diagram of Counter(Left)/Stopwatch(Right)**

Four counters and five stopwatches are implemented on an Arty S7 board. The clock of each counter is connected to each one of the jb_0, jb_2, jb_4, and jb_6 pin inputs. In other words, the counters on the board counts the number of posedges of these inputs. In stopwatches, the five inputs, namely jb_0, jb_2, jb_4, jb_6, and ja_2, are used as the stop inputs. That is, when there is an input, the stopwatch stops. Other inputs and outputs of the counters and stopwatches are connected to inputs and outputs of the sequencer. The pin connections among the modules, along with the output ports, are shown in Figure 7 and Figure 8.

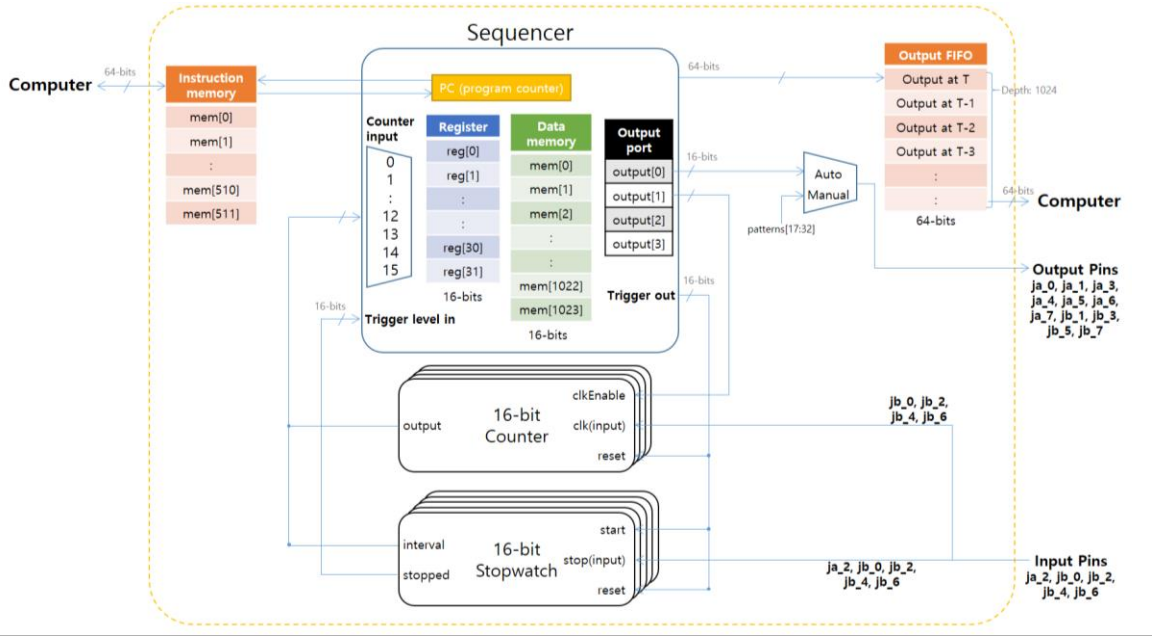


**Figure 7 Sequencer with Peripherals and Input/Output Connections**

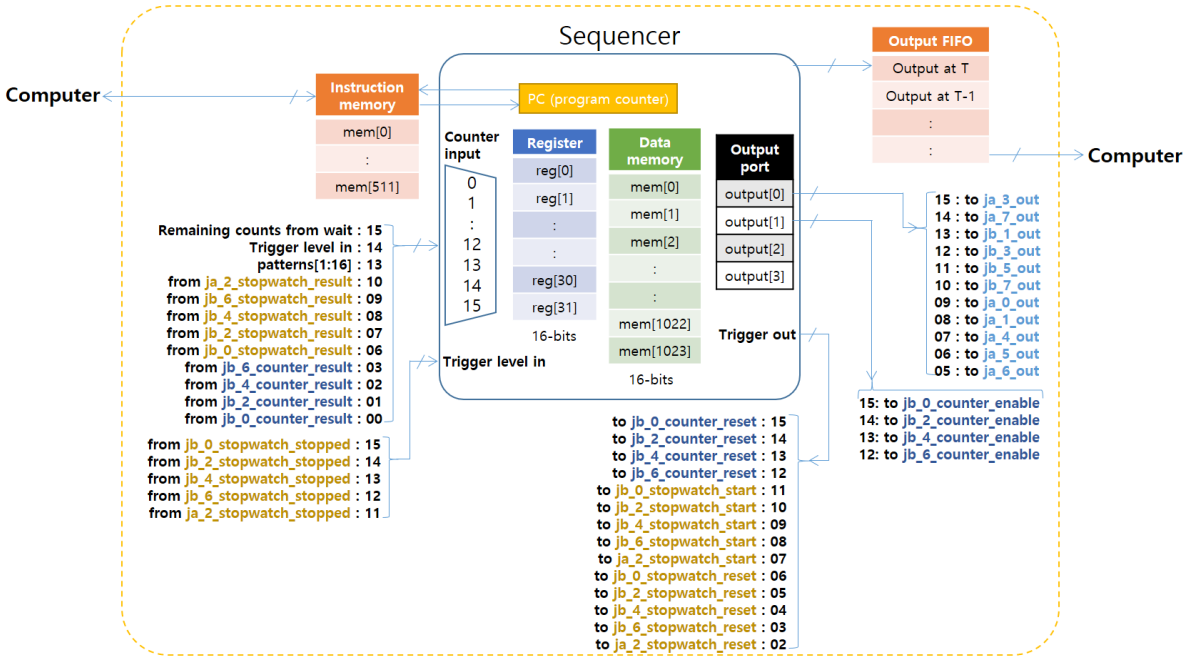


**Figure 8 Sequencer Pin Map (unused pins are neglected)**

The output ports, counters, and stopwatches allow the sequencer to be applied in different experiments. In the following, some examples showing how to utilize these ports and peripherals are introduced with the circuit defined previously.

### 3) Example 4: External Output Port

```
#  1  # Example 4: External Output Port
#  2  s = SequencerProgram()
#  3
#  4  s.set_output_port(hd.external_control_port, [(hd.LED1_out, 1),
      (hd.LED2_out, 0)])
#  5
#  6  s.load_immediate(reg[0], 0)
#  7  s.repeat_wait1 = \
#  8  \
#  9  s.wait_n_clocks(30000, 'Wait 30000 cycles unconditionally')
# 10  s.add(reg[0], reg[0], 1)
# 11  s.branch_if_less_than('repeat_wait1', reg[0], 10000)
# 12
# 13  s.set_output_port(hd.external_control_port, [(hd.LED1_out, 0),
      (hd.LED2_out, 1)])
# 14
# 15  s.load_immediate(reg[0], 0)
# 16  s.repeat_wait2 = \
# 17  \
# 18  s.wait_n_clocks(30000, 'Wait 30000 cycles unconditionally')
# 19  s.add(reg[0], reg[0], 1)
# 20  s.branch_if_less_than('repeat_wait2', reg[0], 10000)
# 21
# 22  s.set_output_port(hd.external_control_port, [(hd.LED2_out, 0)])
# 23  s.write_to_fifo(reg[0], reg[0], reg[0], 10, 'End')
# 24  s.stop()
```

Example 4 shows how to use the external output port of the sequencer. In Line 4, `s.set_output_port()` method is used to change port outputs. The first argument selects which output port will be affected. `hd.external_control_port` in this example indicates that the external output port, or `output[0]` is selected. In the next example, you will see this method controlling a different output port. The method takes a list of tuples as the second argument. A tuple has two entries. The first entry determines the pin and the second entry indicate the wanted output, 0 or 1. Therefore, after the instruction of Line 4 is executed, the LED1 will turn on (`(hd.LED1_out, 1)`) while the LED2 will be off (`(hd.LED2_out, 0)`). Here, note that the aliases (`hd.LED1_out`, `hd.LED2_out`) generated by `HardwareDefinition_type_EX.py` module are used. This is one great advantage of the sequencer in that there is no need to remember the pin port that connects the external devices once you define the mapping in the `HardwareDefinition_type_*.py` module.[3]

Line 6~11 is used to stall the sequencer for approximately 30000 * 10000 = 300 million cycles.

[3] There is an alternative form for `s.set_output_port()`, which uses `bit_pattern` and `mask_pattern` as the second and third argument. However, note that this method cannot directly take the advantage of aliasing. See 'Summary of Instructions' for details.

Recalling that the clock frequency of the sequencer is 100MHz, it stalls the sequencer for approximately 3 seconds. Details of `s.wait_n_clocks()` method will be explained in Example 7.

After 3 seconds of waiting, the port output is again changed (Line 13). This time LED1 is now off (`(hd.LED1_out, 0)`) and LED2 is turned on (`(hd.LED2_out, 1)`). Line 15~20 again stalls the sequencer for approximately 3 seconds. And then finally LED2 is also turned off (`(hd.LED2_out, 0)`) in Line 22. Note that even after the sequencer has stopped by `s.stop()` method, the port output will not automatically be changed to 0. If there were no Line 22 in this example, you will be able see that LED2 is kept on even after 3 seconds of waiting.

#### 4) Example 5: Counter

```
#  1  # Example 5: Counter
#  2  s = SequencerProgram()
#  3
#  4  s.trigger_out([hd.push_button1_counter_reset,
      hd.push_button2_counter_reset], 'Reset counter')
#  5  s.set_output_port(hd.counter_control_port,
      [(hd.push_button1_counter_enable, 1),
      (hd.push_button2_counter_enable, 1), ], 'Start counter')
#  6
#  7  s.load_immediate(reg[0], 0)
#  8  s.repeat_wait = \
#  9  \
# 10  s.wait_n_clocks(50000, 'Wait 50000 cycles unconditionally')
# 11  s.add(reg[0], reg[0], 1)
# 12  s.branch_if_less_than('repeat_wait', reg[0], 10000)
# 13
# 14  s.set_output_port(hd.counter_control_port,
      [(hd.push_button1_counter_enable, 0),
      (hd.push_button2_counter_enable, 0), ], 'Stop counter')
# 15
# 16  s.read_counter(reg[10], hd.push_button1_counter_result)
# 17  s.read_counter(reg[11], hd.push_button2_counter_result)
# 18  s.write_to_fifo(reg[0], reg[10], reg[11], 10, 'Counts within 5s')
# 19
# 20  s.stop()
```

Example 5 shows the method of using counters. Firstly, every counter must be reset in that there is no guarantee that the count is initially set to zero. The reset signals of counters are connected to the trigger outputs of the sequencer. `s.trigger_out()` method outputs one clock pulse through the pins designated in a list, which is the first argument of the method (Line 4).[4]

Then you send enable signals to the counters to start counting. The enable signals are connected

[4] Similar to `s.set_output_port()`, this method also has an alternative form, using `bit_pattern` as the argument. See 'Summary of Instructions' for details.

to `output[1]`, or the counter control output port, so `s.set_output_port()` method is used with the first argument being `hd.counter_control_port` (Line 5). Line 7~12 are used to stall sequencer for approximately 5 seconds. After 5 seconds, the counters stop counting by setting the enables signals to zero (Line 14).

The counter outputs are sent to the sequencer through the counter ports. You can read these values using `s.read_counter()` method. This method loads the value of a counter port (the second argument) in the destination register (the first argument).

Repeat this example with each time clicking push buttons with different numbers of times. The number of clicks should be returned from the sequencer. However, it is highly likely that the data fetched from the sequencer is higher than the number of pushes you have made. This is merely because push buttons are clicked by hand to make the example simple, generating noise signals that have multiple posedges. If delicate experimental setups such as PMTs are used, it will show the exact number of input counts.

Again, note that aliases have made the programming much easier. Provided that `HardwareDefinition_type_EX.py` precisely reflects the pin connection, no consideration on which port is connected to which button is required. Regarding the role of each pin labels, `push_button1(2)_counter_enable` is the enable signal of the first(second) push button and `push_button1(2)_counter_result` is the output of the push button 1(2) counter.

### 5) Example 6: Stopwatch

```
#  1  # Example 6: Stopwatch
#  2  s = SequencerProgram()
#  3
#  4  s.trigger_out([hd.push_button1_stopwatch_reset ,
      hd.push_button2_stopwatch_reset, ], 'Reset stopwatches')
#  5  s.nop() # Between reset and start of the stopwatches , add at least
      one clock
#  6  s.trigger_out([hd.push_button1_stopwatch_start,
      hd.push_button2_stopwatch_start, ], 'Start stopwatches')
#  7
#  8  s.load_immediate(reg[0], 0)
#  9  s.repeat_wait = \
# 10  s.wait_n_clocks(50000, 'Wait 50000 cycles unconditionally')
# 11  s.add(reg[0], reg[0], 1)
# 12  s.branch_if_less_than('repeat_wait', reg[0], 10000)
# 13
# 14  s.read_counter(reg[0], hd.push_button1_stopwatch_result,
      'push_button1 stopwatch result')
# 15  s.read_counter(reg[1], hd.push_button2_stopwatch_result,
      'push_button2 stopwatch result')
# 16  s.read_counter(reg[2], hd.Trigger_level, 'Status of stopwatches')
```

```
# 17
# 18  s.write_to_fifo(reg[0], reg[1], reg[2], 10, 'Stopwatch Example')
# 19
# 20  s.stop()
```

Example 6 shows the method of using stopwatches. In this example, two stopwatches are used, one for each push button. Line 4~6 show how to reset and start stopwatches. First you have to reset the stopwatches in that it may not be set to zero. And then you send the start signal that starts the stopwatches. Note that between the reset and start, there must be at least one clock gap to ensure that the stopwatches act properly. `s.nop()` method is used for generating this gap. This method has no explicit action and just skips one clock. In sending reset and start signals, `s.trigger_out()` method is used because these signals are connected to the trigger output of the sequencer.

Line 8~12 is for waiting the sequencer for approximately 5 seconds.

Line 14~16 loads the results and status of stopwatches in registers. If you run this example without pushing any buttons, you will have `[17042, 17050, 0, 10]` as a result. Note that the results of the two stopwatches (`reg[0]` and `reg[1]`) differ by 8. Recall that the clock of stopwatches are 8 times faster than that of the sequencer. Because the result of push button 2 was read one cycle after the result of push button 1, the value is larger by eight.

Repeat this example several times, sometimes clicking only one of the two buttons and other times clicking both buttons. When you have pushed only push button 1, you will get the result `[xxxxx, 17050, 32768, 10]`. When you have pushed only push button 2, you will get the result `[17042, xxxxx, 16384, 10]`. And finally when you have pushed both buttons, you will get `[xxxxx, yyyyy, 49152, 10]`.`hd.Trigger_level` actually indicates `counter[14]`, which has the trigger-level-in value. Recalling that the trigger-level-in inputs are connected to the stopped signals of stopwatches, reading `hd.Trigger_level` shows which stopwatches have stopped using a 16-bit integer (See '6.2) Outside the Sequencer'). Note that $32768 = 2^{15}$ , $16384 = 2^{14}$ , and 49152=32768+16384. From this, you can see that the 15th(14th) bit (counting from the lowest bit) of `hd.Trigger_level` indicates whether the push button 1(2) has been stopped. The information on which bit shows the stopped signal of which stopwatch is given in Line 39~44 of `HardwareDefinition_v4_01.py` file. You can see that `stopwatch_stopped_TLI` of jb_0(jb_2), the bit assigned for the stopped signal of the push button 1(2), is 15(14). Recalling that you can use aliases for variables related to pins, you can also find this value using the variable `hd.push_button1(2)_stopwatch_stopped_TLI`.

```
# 39  # Trigger_level_in bit assignment
# 40  jb_0_stopwatch_stopped_TLI = 15
# 41  jb_2_stopwatch_stopped_TLI = 14
# 42  jb_4_stopwatch_stopped_TLI = 13
# 43  jb_6_stopwatch_stopped_TLI = 12
```

```
# 44  ja_2_stopwatch_stopped_TLI = 11
```

**HardwareDefinition_v4_01.py Line 39~44(inline comments are deleted)**

One might wonder why the results of push_button1_stopwatch and push_button2_stopwatch should be 17042 and 17050 when they are not stopped. First, using a 16-bit representation of integers, stopwatches can count up to $2^{16} - 1 = 65535$, but the number of stopwatch clocks passed in this example was approximately 50000 * 10000 * 8 cycles, which results in overflows. And there is a timing issue on the `s.wait_n_clocks()` method, which also affects the stopwatch results. This issue will be dealt in the following example.

### 6) Example 7: Timing Issue of Wait

For careful readers, you may have noticed that the document repeatedly used the word "approximately" when `s.wait_n_clocks()` method were used. There are two reasons for this. One is because there were other instructions that uses another one clock, such as `s.add()` or `s.branch_if_less_than()`. The other reason is because `s.wait_n_clocks(n)` method itself does not exactly stall the sequencer execution for `n` clocks. This example shows how to accurately calculate the timing of each instruction and predict stopwatch results.

```
#  1  # Example 7: Timing Issue of Wait
#  2  s = SequencerProgram()
#  3
#  4  s.load_immediate(reg[10], 3)
#  5
#  6  s.trigger_out([hd.push_button1_stopwatch_reset], 'Reset stopwatch')
#  7  s.nop() # Between reset and start, add at least one clock
#  8  s.trigger_out([hd.push_button1_stopwatch_start], 'Start stopwatch')
#  9
# 10  s.read_counter(reg[0], hd.push_button1_stopwatch_result, '1 clock
     after start')
# 11  s.read_counter(reg[1], hd.push_button1_stopwatch_result, '2 clock
     after start')
# 12  s.nop()    # 3 clock after start
# 13  s.read_counter(reg[2], hd.push_button1_stopwatch_result, '4 clock
     after start')
# 14  s.wait_n_clocks(1, 'Wait 1 cycle')
# 15  s.read_counter(reg[3], hd.push_button1_stopwatch_result)
# 16  s.wait_n_clocks(reg[10], 'Wait 3 cycle')
# 17  s.read_counter(reg[4], hd.push_button1_stopwatch_result)
# 18  s.wait_n_clocks(5000, 'Wait 5000 cycle')
# 19  s.read_counter(reg[5], hd.push_button1_stopwatch_result)
# 20
# 21  s.write_to_fifo(reg[0], reg[1], reg[2], 10, 'Timing Issue of Wait')
# 22  s.write_to_fifo(reg[3], reg[4], reg[5], 10, 'Timing Issue of Wait')
# 23
# 24  s.stop()
```

If you run this example, you will get `[10, 18, 34, 10] [74, 130, 40162, 10]`. The first

stopwatch output was read one clock after the stopwatch has been started (Line 10). However, the output value is 10, not 8. This is because the timing of trigger out and counter read are not same in the cycle. The second stopwatch output is read in the next cycle (Line 11). The difference between the first and second stopwatch output is 18-10=8. The third stopwatch output was read after the `s.nop()` method (Line 13). Because of this, there is two cycle difference between the second and third stopwatch read. You can see that the result shows this (34-18=16).

Now, it is natural to think that `s.wait_n_clocks(1)` should be equivalent to `s.nop()` in that both instructions should consume one clock without any particular actions. However this example shows that this is in fact not true. The fourth read of the stopwatch output is done after `s.wait_n_clocks(1)` (Line 15). If `s.wait_n_clocks(1)` was indeed equivalent to `s.nop()`, the result should have been 34+16=50. But the actual result is 74, indicating that `s.wait_n_clocks(1)` in fact have consumed four clock cycles (74–34=40=8*5, one clock is for `s.read_counter()`), or in other words, additional three clocks. You can see that this result is consistent with `s.wait_n_clocks(reg[10])` (`reg[10]` has the value 3) and `s.wait_n_clocks(5000)` too (130-74=56=8*7, 40162-130=40132=8*5004). This is because of the FSM used for wait instructions, but it will not be discussed in detail in this document.

In the example 6, the result when no buttons were pushed was `[17042, 17050, 0, 10]`. This result can be understood accordingly.
((one clock for `s.load_immediate()` + (50000 + 3 additional clocks + 2 clocks for `s.add()` and `s.branch_if_less_than()`) * 10000 iterations + one clock for `s.read_counter()`) * 8 + 2 for timing between start and read) mod ($2^{16}$ for overflow) = ((1+50005*10000+1)*8+2) mod $2^{16}$ = 17042.

### 7) Example 8: Masked operations: (beq_mask, wait_mask)

```
#  1  # Example 8: Masked Operations
#  2  mask = 1 << hd.push_button1_stopwatch_stopped_TLI
#  3  push_buttons_stopped = (1 << hd.push_button1_stopwatch_stopped_TLI)
      + (1 << hd.push_button2_stopwatch_stopped_TLI)
#  4
#  5  s = SequencerProgram()
#  6
#  7  s.trigger_out([hd.push_button1_stopwatch_reset ,
      hd.push_button2_stopwatch_reset, ], 'Reset stopwatches')
#  8  s.nop() # Between reset and start of the stopwatches , add at least
      one clock
#  9  s.trigger_out([hd.push_button1_stopwatch_start,
      hd.push_button2_stopwatch_start, ], 'Start stopwatches')
# 10  s.set_output_port(hd.external_control_port, [(hd.LED1_out, 1)])
# 11
# 12  s.load_immediate(reg[0], 0)
```

```
# 13  s.repeat_wait = \
# 14  s.wait_n_clocks_or_masked_trigger(50000,
      [(hd.push_button1_stopwatch_stopped_TLI, 1)])
# 15  s.read_counter(reg[1], hd.Trigger_level, 'Status of stopwatches')
# 16  s.branch_if_equal_with_mask('turn_off_LED', reg[1],
      push_buttons_stopped, mask)
# 17  s.add(reg[0], reg[0], 1)
# 18  s.branch_if_less_than('repeat_wait', reg[0], 10000)
# 19
# 20  s.turn_off_LED = \
# 21  s.set_output_port(hd.external_control_port, [(hd.LED1_out, 0)])
# 22
# 23  s.write_to_fifo(reg[0], reg[1], reg[1], 10, 'Masked Operations')
# 24
# 25  s.stop()
```

For branch_if_equal and wait_n_clocks operations, there are masked versions of them. Example 7 shows the usage of these operations. In Line 7~10, stopwatches for push button 1 and push button 2 are started and LED1 is turned on. If you run this example without pushing any buttons, the LED will be on for about 5 seconds and turn off. If you push the push button 2 only, the action will be the same. That is, the LED will still be on for about 5 seconds and turn off. Now if you run this example and push the push button 1, the LED will turn off right away and the sequencer will stop (regardless of whether you have pushed the push button 2). This is because of the methods used in Line 14~16.

`s.wait_n_clocks_or_masked_trigger()` method is a wait instruction that waits for specified number of clocks (either by integer value or register value given as the first argument). However, one thing that is different with `s.wait_n_clocks()` is that while `s.wait_n_clocks()` is an unconditional wait instruction, `s.wait_n_clocks_or_masked_trigger()` stops waiting when certain condition is satisfied. This condition is given as the second argument of the method. In the example, the second argument is a list of tuples, with each tuple having two entries, one being a trigger-level-in input and the other being the target value. Among 16 trigger-level-in input bits, the sequencer compares only the designated trigger level inputs, and if they all are identical with the corresponding target value, the sequencer stops waiting and proceed to the next instruction. In this example, the second argument is given as `[(hd.push_button1_stopwatch_stopped_TLI, 1)]`. Once you push the push button 1, the stopwatch of push button 1 stops and `hd.push_button1_stopwatch_stopped_TLI` becomes 1. Then since the condition is satisfied, the sequencer stops waiting and executes the next instruction. The condition can also be given as separate two arguments with the former specifying the target bit pattern and the latter specifying the bitwise mask pattern. Therefore in this example, changing Line 14 to `s.wait_n_clocks_or_masked_trigger(50000, 0x8000, 0x8000)` will have the same action.

`s.branch_if_equal_with_mask()` method is similar to `s.branch_if_equal()` method but compares only the bits that are represented by 1 in the binary representation of the mask (the

fourth argument). In the example, mask is `1 << 15 = 0x8000` (Line 2) (recall from Example 6 that `push_button1_stopwatch_stopped_TLI` is 15)[5], so the sequencer compares only the highest bit of `reg[1]` and `push_buttons_stopped`, the variable having the two highest bits as 1 (Line 3). Therefore if you press the push button 1, the condition is satisfied and the sequencer jumps to `s.turn_off_LED`. And even if `push_buttons_stopped` has 1 on the second highest bit, pressing the push button 2 has no effect on branching since the second highest bit of the mask is zero.

## 7. Summary of Instructions

Having introduced all the instructions of the sequencer, a brief summary or list of the instructions are provided in this section. This list will be helpful when you are in the actual procedure of programming. (All methods here can have one additional argument `comment` as a string.)

#### 1) Load Value into a Register (Initialize a Register)

```
load_immediate(reg[n], value)
-   reg[n] <= value
```

- used for initializing a register

#### 2) Arithmetic Operations

```
add(reg[n], reg[m], reg[l])
-   reg[n] <= reg[m] + reg[l]

add(reg[n], reg[m], value)
-   reg[n] <= reg[m] + value

subtract(reg[n], reg[m], reg[l])
-   reg[n] <= reg[m] - reg[l]

subtract(reg[n], reg[m], value)
-   reg[n] <= reg[m] - value
```

#### 3) Branch/Jump Operations

* `addr` can be either a string, an integer, or a class attribute, but it is highly recommended to use the string data type.

---

[5] Note that whether `push_button1_stopwatch_stopped_TLI` is 15 or not is in fact not important on whether the sequencer will act correctly or not as long as you have used the proper aliases having the same header '`push_button1_`'.

`branch_if_less_than(addr, reg[n], reg[m])`

- `if (reg[n] < reg[m]) nextPC <= addr; else nextPC <= PC + 1`

`branch_if_less_than(addr, reg[n], value)`

- `if (reg[n] < value) nextPC <= addr; else nextPC <= PC + 1`

`branch_if_equal_with_mask(addr, reg[n], reg[m], mask)`

- `if ((reg[n] & mask) == (reg[m] & mask)) nextPC <= addr;`
  `else nextPC <= PC + 1`
- Only compare bit locations where the value is 1 in the binary representation of `mask`

`branch_if_equal_with_mask(addr, reg[n], value, mask)`

- `if ((reg[n] & mask) == (value & mask)) nextPC <= addr;`
  `else nextPC <= PC + 1`
- Only compare bit locations where the value is 1 in the binary representation of `mask`

`branch_if_equal (addr, reg[n], reg[m])`

- `if (reg[n] == reg[m]) nextPC <= addr; else nextPC <= PC + 1`

`branch_if_equal(addr, reg[n], value)`

- `if (reg[n] == value) nextPC <= addr; else nextPC <= PC + 1`

`jump(addr)`

- Unconditionally jump to `addr`

#### 4) Wait Operations

* `list_of_bit_position_and_value` is a list of tuples. Each tuple has `bit_position` (0~15) and `bit_value` (0 or 1) as entries. When this argument is used, `bit_pattern` has 1s in `bit_position`s and 0s in other locations, and `bit_mask` has `bit_value` in `bit_position`s and 0s in other locations.

`wait_n_clocks_or_masked_trigger(reg[m], bit_pattern, bit_mask)`

- Wait for `reg[m]` clocks.
  `if ((trigger_level_in & bit_mask) == (bit_pattern & bit_mask)) abort wait`
- Only compare bit locations where the value is 1 in the binary representation of `bit_mask`

`wait_n_clocks_or_masked_trigger(reg[m], list_of_bit_position_and_value)`

- Wait for `reg[m]` clocks.
  `if ((trigger_level_in & bit_mask) == (bit_pattern & bit_mask)) abort wait`
- `bit_pattern` and `bit_mask` are generated from `list_of_bit_position_and_value`.

`wait_n_clocks_or_masked_trigger(clock_count, bit_pattern, bit_mask)`

- Wait for `clock_count` clocks.
  `if ((trigger_level_in & bit_mask) == (bit_pattern & bit_mask)) abort wait`
- Only compare bit locations where the value is 1 in the binary representation of `bit_mask`

`wait_n_clocks_or_masked_trigger(clock_count, list_of_bit_position_and_value)`

- Wait for `clock_count` clocks.
  `if ((trigger_level_in & bit_mask) == (bit_pattern & bit_mask)) abort wait`
- `bit_pattern` and `bit_mask` are generated from `list_of_bit_position_and_value`.

`wait_n_clocks(reg[m])`

- Wait for `reg[m]` clocks unconditionally

`wait_n_clocks (clock_count)`

- Wait for `clock_count` clocks unconditionally

### 5) Using Peripherals and External Devices

`set_output_port(port_number, bit_pattern, mask_pattern)`

- `port[port_number] <= ((bit_pattern & mask_pattern) | (port[port_number] & ~mask_pattern))`
- Change the selected output port (bit location where the value is 1 in the binary representation of `mask_pattern`) according to the `bit_pattern`

`set_output_port(port_number, list_of_bit_position_and_value)`

- `port[port_number] <= ((bit_pattern & mask_pattern) | (port[port_number] & ~mask_pattern))`
- Change the selected output port (bit location where the value is 1 in the binary representation of `mask_pattern`) according to the `bit_pattern`
- `list_of_bit_position_and_value` is a list of tuples., Each tuple has `bit_position` (0~15) and `bit_value` (0 or 1) as entries. `bit_pattern` has 1s in `bit_position`s and 0s in other locations, and `mask_pattern` has `bit_value` in `bit_position`s and 0s in other locations.

`trigger_out(bit_pattern)`

- Generate a single cycle pulse of `bit_pattern`
- Pulse is outputted through the trigger output

`trigger_out(list_of_bit_position)`

- `list_of_bit_position` is a list of integers (0~15)
- Generate a single cycle pulse on ports mentioned in `list_of_bit_position`
- Pulse is outputted through the trigger output

`read_counter(reg[n], counter_index)`

- `reg[n] <= input[counter_index]`
- Read the value of the counter port and load it to the register

### 6) Memory Access

`load_word_from_memory(reg[target], reg[m])`

- `reg[target] <= data_memory[reg[m]]`

`store_word_to_memory(reg[m], reg[source])`

- `data_memory[reg[m]] <= reg[source]`

### 7) Send Data to PC

`write_to_fifo(reg[n], reg[m], reg[l], value)`

- Input 64-bit string `(reg[n], reg[m], reg[l], value)` as an entry of the output FIFO

### 8) Stop

`stop()`

- Stop the sequencer

### 9) No operation

`nop()`

- No operation (skips one clock)