# Overall design

## Memory

- Instruction memory: 512 x 48-bits

- Registers: 32 x 16-bits

- Data memory: 1024 x 16-bits

## Inputs

- Input ports: (16 ports) x (16-bits)

- Input trigger bits: 16-bits

## Outputs

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

- Output trigger bits: 16-bits

## Data transfer to computer

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

## Speed

- Clock speed: 100 MHz

- Single-cycle processor: execution of each instruction takes only 1-clock except wait instruction

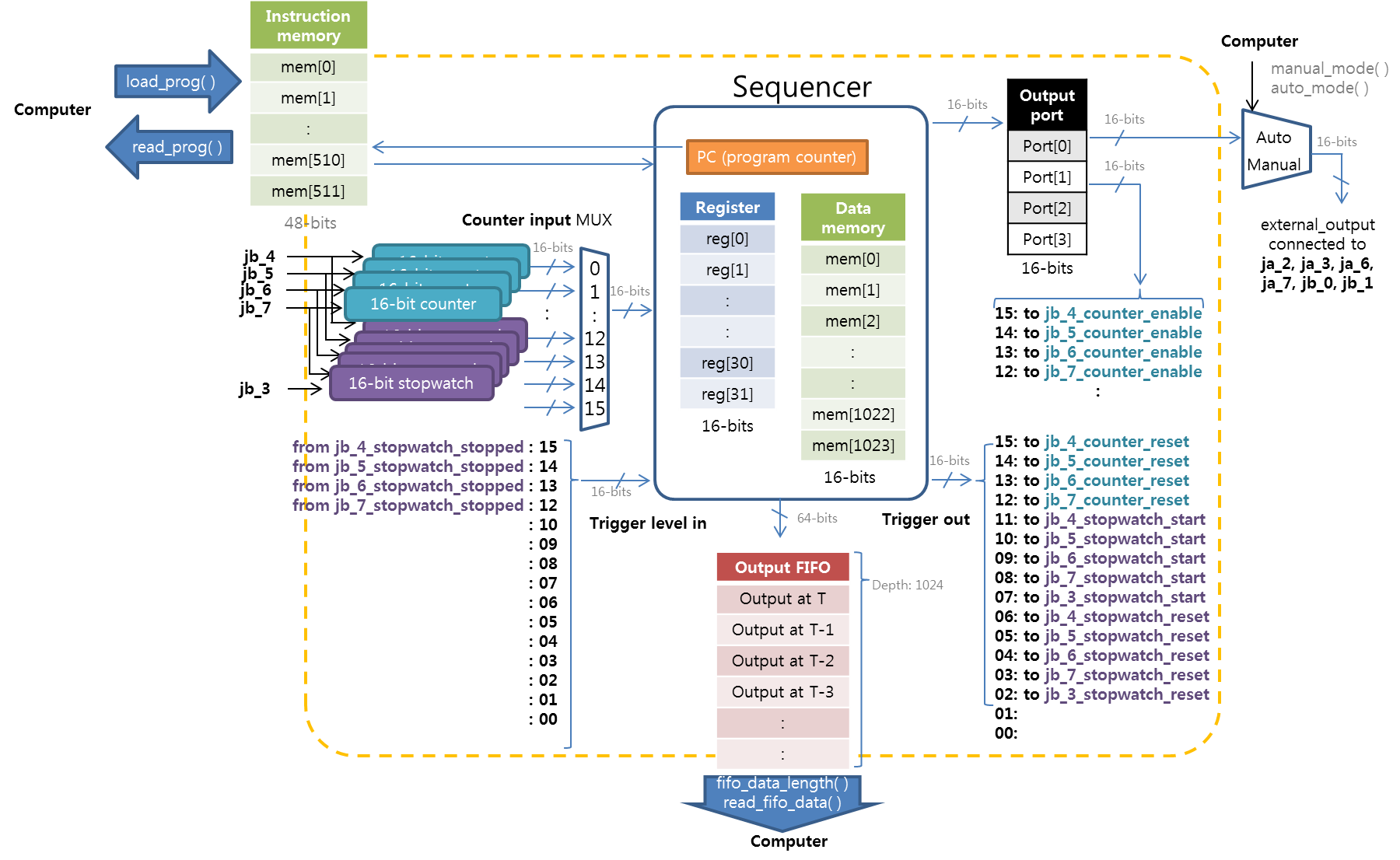
- Stopwatch clock speed: 800 MHz

## Peripherals

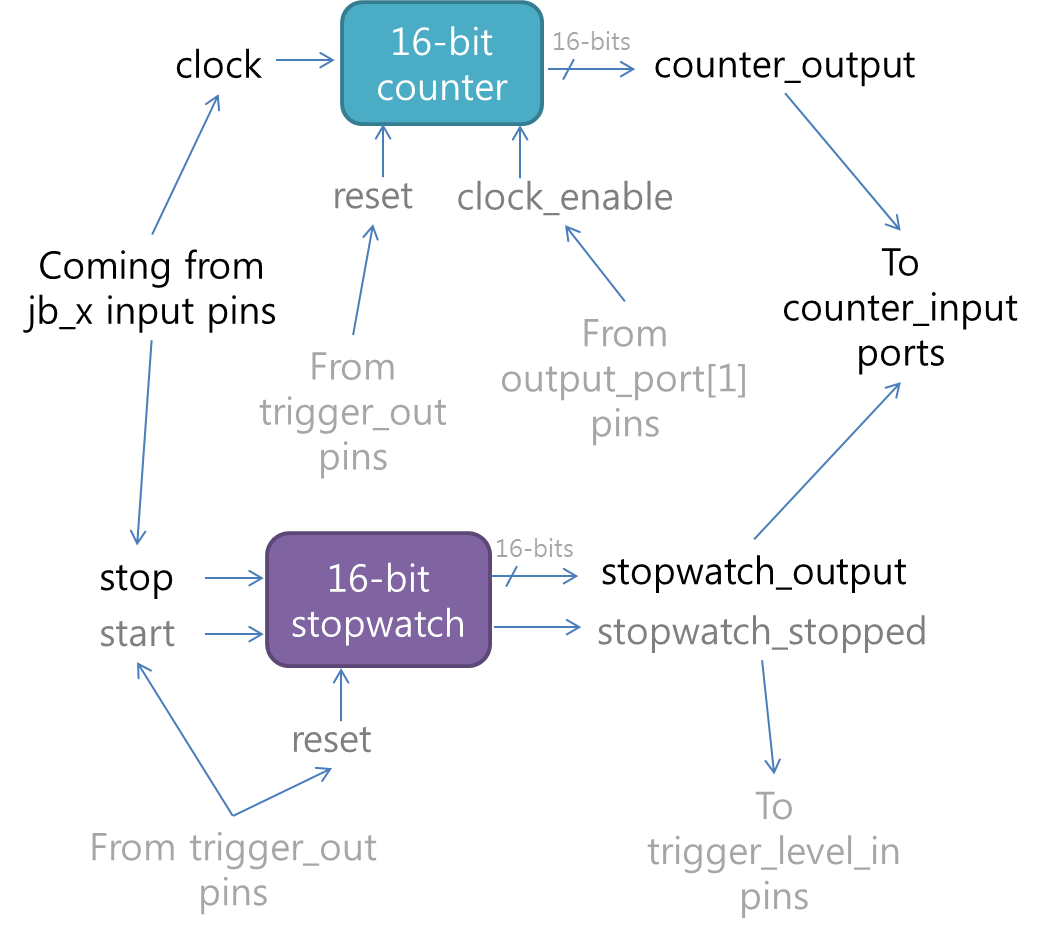
- Input counters: 4 x 16-bit

- Stopwatches: 5 x 16-bit

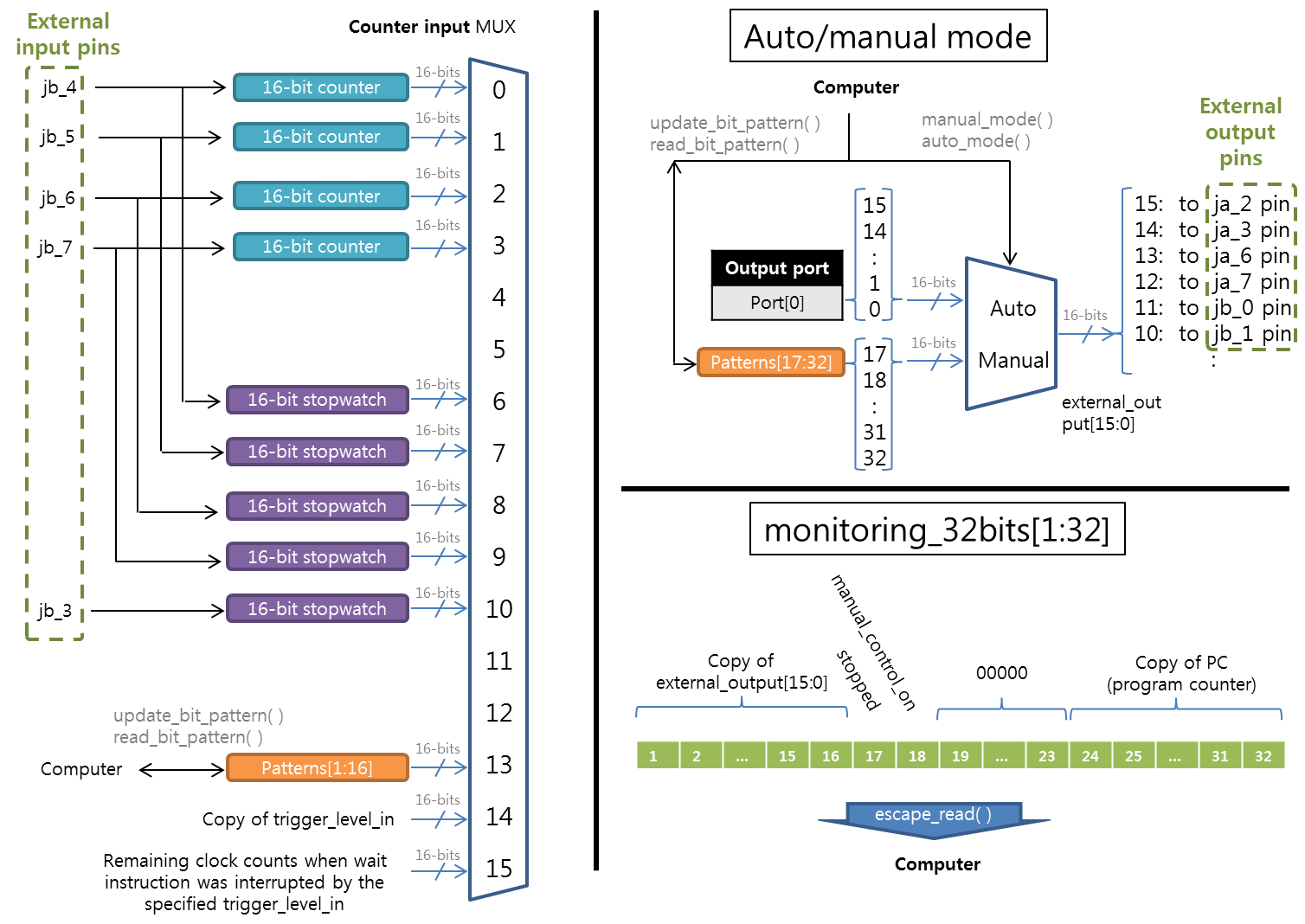
## Explanation of components



* The system is composed of three types of components similar to the modern microprocessor: memory, input, output
* There are three types of memory in this design
  + Instruction memory is where the program is stored. Program can be downloaded from computer using load\_prog( ) function and the stored program can be read back to the computer using read\_prog( ) function
  + Registers are the temporary storage space. The data in registers are generally used for calculations and comparison, and for input operation, it is used as destination of the input value. For data memory operation, it contains both the memory address and the data. In the current design, registers are not directly used for output operations except output FIFO, because individual output pins are generally controlled in a fixed pattern. At the program start, registers are NOT automatically reset, so do not assume that the initial value of the registers will be zero.
  + Data memory is a larger storage space. Compared to registers, the address of data memory can be changed during execution, so array operation is possible. At the program start, data memory is NOT automatically reset, so you should initialize the data memory at the beginning if you want to use the data memory. On the other hand, you can utilize such kind of situation to combine the new data with the previous data.
* There are two types of input in this design
  + Currently there are sixteen counter input ports. Each counter input port is 16-bit-wide, so it is useful for reading multiple bits such as the result of counter or stopwatches, and it can be used for other future purposes. This port is also used to read multiple individual bits such as stopped bits of multiple stopwatches. Once 16-individual bits are read into register, individual bits can be tested by using branch\_if\_equal\_with mask.
  + There are 16 input bits for trigger\_level\_in. The main purpose of trigger\_level\_in is to abort wait instruction before the specified period if the desired conditions are satisfied. trigger\_level\_in data is also available through counter input port 14.
* There are two types of output in this design
  + Currently there are four output ports. Each output port is 16-bit wide, but by using mask bits, individual bits can be changed, so it can be used for both parallel output and individual outputs. In the current design, pins from output port[0] are connected to the external pins, and pins from output port[1] are connected to control internal components.
  + There are 16 trigger output bits. These pins are used to generate single-clock pulses with trigger\_out instruction. Currently they are connected to the reset inputs of counters and both reset and start inputs of stopwatches. In principle, output\_port can be also used to generate single-clock pulse by turning on and off the target bits, but this requires two lines of instruction, so when coding efficiency is critical, trigger output bits are recommended.
* To transfer any information to the computer, Output FIFO is available. “write” instruction writes (reg[n], reg[m], reg[l], value) to the FIFO. “value” field can be any constant 16-bit value and they can be generally used to label the content. For example, when we write PMT counts obtained during Doppler cooling/initialization, we can label them with constant value 10. Then, when we write coincidence time, we can label them with constant value 20. These labels can also be used as error code. Output FIFO can be considered as console output in the modern operating system.
* Currently there are two types of peripherals, and more peripherals can be implemented in FPGA, if necessary.
  + Counter is generally used to measure the PMT counts when the clock\_enable signal of the counter is high. The counter value won’t be reset until the explicit reset pulse is provided, and the counter results are generally read by “read” instruction through counter input ports.

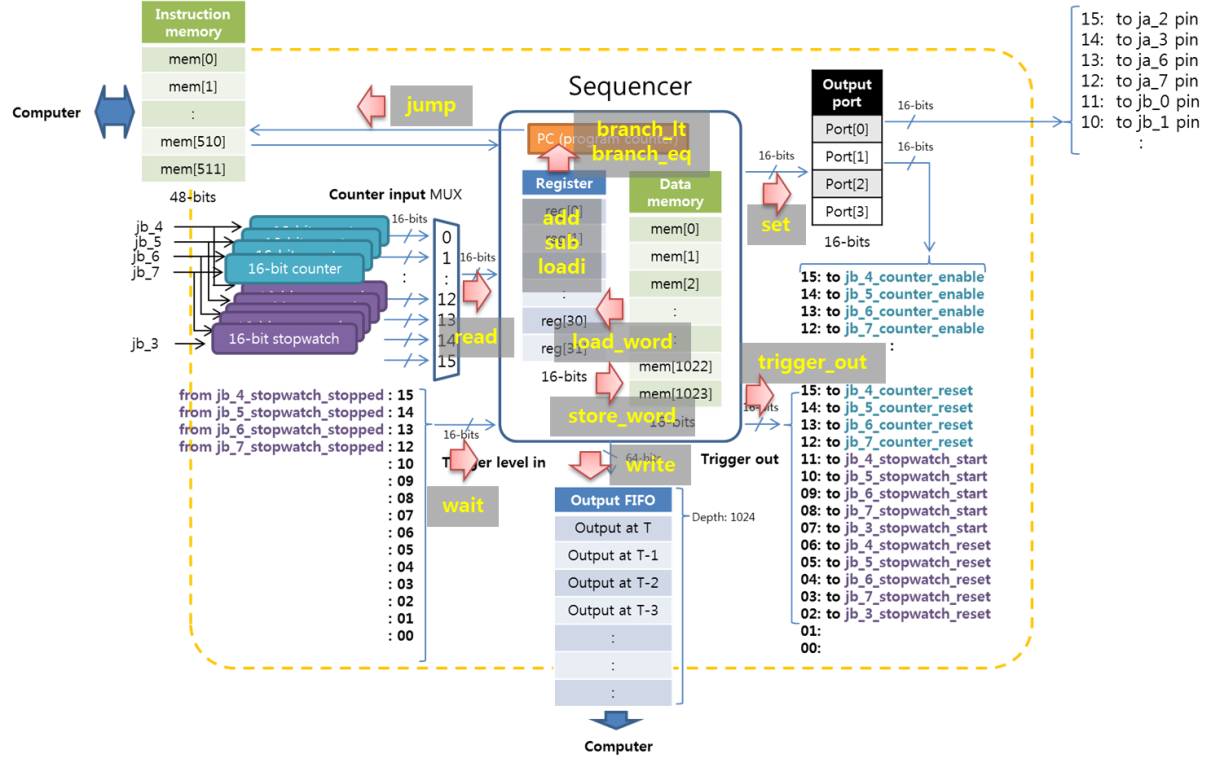


* + Stopwatch
    - Stopwatch measures the timing of the rising edge of the input signal. Different from usual expectation, the start pin of the stopwatch is not connected externally. Therefore it should be always started internally, and the external signal can only stop the stopwatch. Even though this type of design might require more stopwatch modules than usual, this approach has an advantage that we can measure the response time w.r.t. the internal timing, and also compare the timing of the multiple signals. Of course, a single input can be used as the start of the multiple stopwatches, but in this case, we have to worry about the impedance matching for multiple fan-outs.
    - Stopwatch always should be reset before it will be used. Stop signal occurring between the reset and start signal will be ignored. Once the rising edge of the start signal has arrived, it will count the number of 800MHz clocks (1.25 ns period), and the counting will stop after the rising edge of the stop signal has arrived. Once the stopwatch has stopped, the output value won’t be changed until it will be reset.
    - You should add at least single nop instruction or some other instructions between stopwatch reset and the start, because in some stopwatches, there is a race condition between the reset signal and the start signal. If the rising edge of the start signal arrives before the falling edge of the reset signal, the stopwatch won’t be started by the given start signal.



## Summary of instruction set

* Register change and arithmetic
  + add: add register and/or value
  + sub: subtract register and/or value
  + loadi: fill the register with some value
* Input
  + read: fill the register with a value available at the input port
* Output
  + set: change the output bits of output port
  + trigger\_out: generate a single-clock pulse
* Jump
  + blt: if some register is less than another register or a value, jump to other location
  + beq: if masked bits of some register is equal to another register or a value, jump to other location
  + jump: jump to a specific address
* Data exchange with memory
  + load\_word: read a value stored at a specific address of data memory and fill the destination register
  + store\_word: fill the content of the data memory at specific address with the value in a source register
* Others
  + write: send data to Output FIFO
  + wait: wait for specified clocks or abort if the specified trigger input is detected
  + nop: no operation (consumes one clock)
  + stop: stop



# Instruction set

## Instruction format (6-bytes)

{opcode (8-bits), r1 (8-bits), r2 (8-bits), r3 (8-bits), imm1 (16-bits), imm2 (16-bits)}

{opcode (8-bits), r1 (8-bits), imm3 (16-bits), imm1 (16-bits), imm2 (16-bits)}

Generally each field has its frequent usage with a few exceptions:

* r1: most versatile register. Used for most of instructions.
* r2: used for most of instructions.
* r3: only this register can represent the target registers of any instruction. Therefore it will be the destination of add, subtract, loadi, read, load\_word commands.
* imm1: immediate value operand for most of commands such as add, subtract, loadi, index for counter input, and bit patterns for output ports, trigger\_output, and wait commands. But for beq, imm1 is used as mask bit rather than operand.
* imm2: destination address for branch and jump commands. Mask bits for output ports, and wait commands.

imm3: this value is used as immediate value of blti (branch if less than immediate value) and beqi (branch if equal to immediate value).

## Add registers

* add (reg[n], reg[m], reg [l])
* reg[n] <= reg[m] + reg[l]
* {add, r1 (m), r2 (l), r3 (n), imm1 (0), imm2 (0)}
* add (reg[n], reg[m], value)
* reg[n] <= reg[m] + value
* {addi, r1 (m), r2 (0), r3 (n), imm1 (value), imm2 (0)}

## Subtract registers

* subtract (reg[n], reg[m], reg [l])
* reg[n] <= reg[m] - reg[l]
* {sub, r1 (m), r2 (l), r3 (n), imm1 (0), imm2 (0)}
* subtract (reg[n], reg[m], value)
* reg[n] <= reg[m] – value
* {subi, r1 (m), r2 (0), r3 (n), imm1 (value), imm2 (0)}

## Load a value into a register

* load\_immediate (reg[n], value)
* reg[n] <= value
* {loadi, r1 (0), r2 (0), r3 (n), imm1 (value), imm2 (0)}
* Copy of one register to another register can be done by adding 0 to a register

## Read a value from a selected counter into a register

* read\_counter (reg[n], counter\_index)
* reg[n] <= counter\_input[counter\_index]
* {read, r1(0), r2(0), r3(n), imm1(counter\_index), imm2(0)}

## Set output bits of a selected output\_port

* set\_output\_port (port\_number, bit\_pattern, mask\_pattern)
* Output bits of port [port\_number] will be updated according to (bit\_pattern, mask\_pattern)
* {set, r1(0), r2(0), r3(port\_number), imm1 (bit pattern), imm2 (mask\_pattern)}

Note: In the instruction code, port\_number is stored in the index of r3, but this does not mean that the value stored in r3[port\_number] register will be used for port\_number. Immediate value of port\_number will be used as port number. In the future architecture, imm3 should be used instead of r3.

## Write three registers to a FIFO

* write\_to\_fifo (reg[n], reg[m], reg[l], value)
* FIFO <= (reg[n], reg[m], reg[l], value)
* {write, r1(n), r2(m), r3(l), imm1(value), imm2(0)}

## Branch to address if reg[n] is less than (reg[m] or value)

* branch\_if\_less\_than (addr, reg[n], reg[m])
* If (reg[n] < reg[m]), go to addr. If not, move to the next instruction.
* {blt, r1(n), r2(m), r3(0), imm1(0), imm2(addr)}
* branch\_if\_less\_than (addr, reg[n], value)
* If (reg[n] < value), go to addr. If not, move to the next instruction.
* {blt, r1(n), imm3(value), imm1(0), imm2(addr)}

## Branch to address if reg[n] is equal to (reg[m] or value) with mask

* branch\_if\_equal\_with\_mask (addr, reg[n], reg[m], mask)
* If (reg[n] == reg[m] for masked bits), go to addr. If not, move to the next instruction.
* {beq, r1(n), r2(m), r3(0), imm1(mask), imm2(addr)}
* branch\_if\_equal\_with\_mask (addr, reg[n], value, mask)
* If (reg[n] == value for masked bits), go to addr. If not, move to the next instruction.
* {beqi, r1(n), imm3(value), imm1(mask), imm2(addr)}
* branch\_if\_equal (addr, reg[n], reg[m])
* If (reg[n] == reg[m]), go to addr. If not, move to the next instruction.
* {beq, r1(n), r2(m), r3(0), imm1(0), imm2(addr)}
* branch\_if\_equal (addr, reg[n], value)
* If (reg[n] == value), go to addr. If not, move to the next instruction.
* {beqi, r1(n), imm3(value), imm1(0), imm2(addr)}

## Jump to address

* jump (addr)
* Jump to address
* {jump, r1(0), r2(0), r3(0), imm1(0), imm2(addr)}

## Wait for (clock\_count or reg[m]) clocks w/ (some masked bit pattern) trigger

* wait\_n\_clocks\_or\_masked\_trigger (reg[m], bit\_pattern, bit\_mask)
* Wait for reg[m] clocks. During this period, if the masked bits of trigger\_level\_in matches the given bit\_pattern, abort wait and proceed to the next instruction.
* {wait, r1(m), r2(0), r3(0), imm1(bit\_pattern), imm2(bit\_mask)}
* wait\_n\_clocks\_or\_masked\_trigger ( clock\_count, bit\_pattern, bit\_mask)
* Wait for clock\_count clocks. During this period, if the masked bits of trigger\_level\_in matches the given bit\_pattern, abort wait and proceed to the next instruction.
* {waiti, r1(0), imm3(clock\_count), imm1(bit\_pattern), imm2(bit\_mask)}
* wait\_n\_clocks (clock\_count or reg[m])
* Wait unconditionally.

Note: if mask == 0, it will be unconditional waiting. When the wait is interrupted by the pattern trigger, the remaining counts will be available through counter[15]. If the timer expires without the pattern trigger, counter[15] bit will be reset.

## Load 16-bit word from memory address to a target register

* load\_word\_from\_memory (reg[target], reg[m])
* reg[target] <= (data in data\_memory at address (reg[m]) )
* {load\_word, r1(m), r2(0), r3(target), imm1(0), imm2(0)}

## Store 16-bit word in source register to a memory address

* store\_word\_to\_memory (reg[m] , reg[source])
* data\_memory at address (reg[m]) <= reg[source]
* {store\_word, r1(m), r2(0), r3(source) , imm1(0), imm2(0)}

## Generate a pulse in the specified bits

* trigger\_out (bit\_pattern)
* Generate a single cycle pulse in the specified bits
* {trigger\_out, r1(0), r2(0), r3(0), imm1(bit\_pattern), imm2(0)}

## Stop

* stop ( )
* Stops the sequence running
* {stop, r1(0), r2(0), r3(0), imm1(0), imm2(0)}

## No operation

* Nop ( )
* This instruction doesn’t do anything. It is useful for waiting for single cycle.
* {nop, r1(0), r2(0), r3(0), imm1(0), imm2(0)}

# Software structure

## ArtyS7 class in ArtyS7\_v1\_01.py

This module implements basic communication protocol between PC and FPGA. FPGA can be programmed either by Vivado or Digilent tool. If the FPGA binary file is written to a flash memory on Arty board, it will be automatically downloaded every time its power is turned on.

ArtyS7 module supports two types of normal packets:

* send\_command( ) method sends “! <byte\_count><ASCII\_string>\r\n” string to FPGA. Maximum length of ASCII string can be 15.
* send\_mod\_BTF\_string( ) method sends “#<num\_digit><byte\_count><raw\_data>\r\n” string to FPGA. BTF (Binary Transfer Format IEEE 488.2 # format) only uses decimal characters for both <num\_digit> and <byte\_count>. Here I use hexadecimal characters, and called it as modified BTF.

To abort the normal packets in the middle of transfer, escape sequence can be used. If data\_receiver in FPGA detects “\x10” character (DLE (Data Link Escape) character in ASICC table, decimal value 16) and it is not followed by additional “\x10” character, then it is considered as escape sequence. If “\x10\x10” sequence is detected, it will be received as single “\x10” value. There are following escape sequences:

* “\x10C”: This is a special escape sequence to clear the input buffer. If this command is encountered, the transmitter buffer will be also cleared and FPGA will send “\x10C” back to PC . FSM will reset to the IDLE state.
* “\x10R”: This is a special escape sequence to read the current status of 32 bits. If this command is encountered, the transmitter buffer will be also cleared and FPGA will send “\x10R” + <monitoring\_32bits> + “\r\n” and FSM will reset to the IDLE state.

The following escape sequences are meaningful when waveform capture function is implemented in FPGA. This is implemented only for the debugging purpose, so with normal bit file, it won’t be implemented.

* “\x10T”: This is a special escape sequence to set trigger setting.
* “\x10A”: This is a special escape sequence to arm trigger.
* “\x10W”: This is a special escape sequence to read the captured waveform date. If this command is encountered, the transmitter buffer will be also cleared and send “\x10W” and the waveform data will be transferred to PC in the form of BTF. FSM will reset to the IDLE state.

### Low-level communication methods

* send\_command(self, cmd): sends CMD-type message
* send\_mod\_BTF\_string(self, modified\_BTF): sends BTF-type message given as string
* send\_mod\_BTF\_int\_list(self, modified\_BTF): sends BTF-type message given as list of bytes
* read\_next(self): reads the next character and return it. If “\x10” is encountered, it will be handled either as escape character or normal “\x10”
* flush\_input(self): flush all the data in the read buffer
* read\_next\_message(self, escape\_debug=False): This will read normal CMD or BTF types. If escape sequence is encountered, the escapeSequenceDetected exception will be raised.

### Sending escape sequence and waveform capture

* escape\_reset(self): send “\x10C” and read “\x10C”
* escape\_read(self): send “\x10R” and read “\x10R” message with monitoring\_32bits[1:32]
* check\_waveform\_capture(self): check whether waveform capture module is implemented in FPGA
* read\_captured\_waveform(self, print\_output=False): read captured waveform data

### Update and read patterns[1:32]

* update\_bit\_pattern(self, list\_of\_bit\_position\_and\_value): updates the bits as given in the list
* read\_bit\_pattern(self): reads the current values in patterns[1:32]

### Sequencer control

* load\_prog(self, addr, prog): writes single line of instruction to the specified address
* read\_prog(self, addr): reads the stored instruction in the specified address
* start\_sequencer(self): starts the execution of the program in the instruction memory
* manual\_mode(self): external output pins will be controlled by patterns[17:32]
* auto\_mode(self): external output pins will be controlled by output\_port[0] of the sequencer
* sequencer\_running\_status(self): print out the current running status obtained by monitoring\_32bits
* fifo\_data\_length(self): checks the amount of data in the output FIFO
* read\_fifo\_data(self, length): reads the data in the output FIFO

## SequencerProgram class in SequencerProgram\_v1\_07.py

This module is used to make a program for the sequencer. SequencerProgram class is used for the following three purposes:

* SequencerProgram class keeps a program\_list internally, and as the method corresponding to the instruction is called, that instruction is added to the program\_list.
* When program( ) method is called without target argument specified, it will show the current sequencer program stored in program\_list. This is useful for debugging.
* When program( ) method is called with target argument, it will actually download the program in program\_list to the instruction memory of the sequencer.

To write a program, start with SequencerProgram class object such as

s = SequencerProgram( )

Then as you would write a program, just call a method of the s object like

s.load\_immediate(reg[1], 3)

The above instruction means that value “3” will be stored in a register with index [1]. When the sequencer’s memory is programmed, this instruction will be stored at address 0 of instruction memory.

s.add(reg[3], reg[1], 10)

The above instruction means that value “10” will be added to a value stored in a reg[1] (value “3”) and the result (“13”) will be stored in reg[3]. When the sequencer’s memory is programmed, this instruction will be stored at address 1 of instruction memory.

### Summary of the methods corresponding to instructions

* Register change and arithmetic
  + add (reg[n], reg[m], reg [l])
  + add (reg[n], reg[m], value)
  + subtract (reg[n], reg[m], reg [l])
  + subtract (reg[n], reg[m], value)
  + load\_immediate (reg[n], value)
* Input
  + read\_counter (reg[n], counter\_index)
* Output
  + set\_output\_port (port\_number, bit\_pattern, mask\_pattern)
  + trigger\_out (bit\_pattern)
* Jump
  + branch\_if\_less\_than (addr, reg[n], reg[m])
  + branch\_if\_less\_than (addr, reg[n], value)
  + branch\_if\_equal\_with\_mask (addr, reg[n], reg[m], mask)
  + branch\_if\_equal\_with\_mask (addr, reg[n], value, mask)
  + branch\_if\_equal (addr, reg[n], reg[m])
  + branch\_if\_equal (addr, reg[n], value)
  + jump (addr)
* Data exchange with memory
  + load\_word\_from\_memory (reg[target], reg[m])
  + store\_word\_to\_memory (reg[m] , reg[source])
* Others
  + write\_to\_fifo (reg[n], reg[m], reg[l], value)
  + wait\_n\_clocks (reg[m])
  + wait\_n\_clocks\_or\_masked\_trigger (reg[m], bit\_pattern, bit\_mask)
  + wait\_n\_clocks (clock\_count)
  + wait\_n\_clocks\_or\_masked\_trigger ( clock\_count, bit\_pattern, bit\_mask)
  + nop ( )
  + stop ( )

Once the program\_list is made, it can be reviewed by calling program( ) method. program( ) method has the following optional arguments:

program(self, show=True, show\_comment=True, target=None, machine\_code=False, hex\_file=None)

* show: if True, program( ) method will interpret each instruction code and show them in human-friendly message.
* show\_comment: if True, program comment will be shown together.
* target: if target argument is specified, the program will be loaded into the target’s instruction memory. Target should be ArtyS7 class object.
* machine\_code: if True, raw value of the instruction code will be shown. This is useful especially for simulation with Vivado.
* hex\_file: if file name with a path is specified, memory file for Vivado simulation will be generated.

## Operation

Once the program is loaded into the instruction memory, program can be run by calling start\_sequencer( ) method of the ArtyS7 object. Default mode of the sequencer is “Manual” mode, so make sure to call auto\_mode( ) method at least once.

At the start, only PC will be reset. Other memory types such as register file or data memory won't be reset.

The running status can be checked by sequencer\_running\_status( ) method of ArtyS7 class object.

To check the amount of data in output FIFO, call fifo\_data\_length( ) method of ArtyS7 class object.

To read data in output FIFO, call read\_fifo\_data(length) method of ArtyS7 class object.

# Debugging

## Simulator

* If there is any request for a simulator, I will write a software simulator.

## Debugging with hardware simulator in Vivado

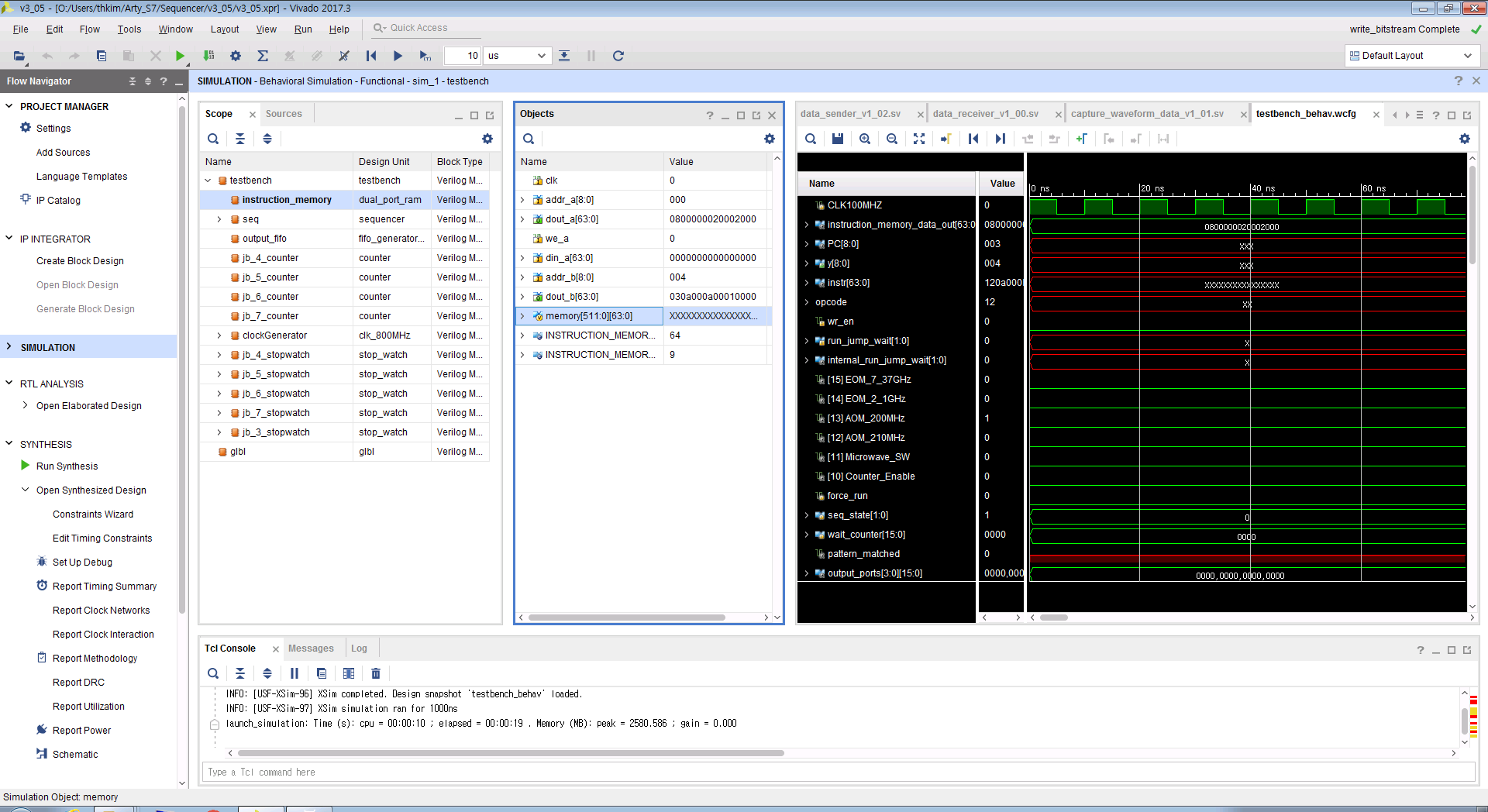
To simulate the running status of the sequencer program, Vivado simulator can be used. Currently “Simulation Sources / sim\_1 / Memory File / testbench.sv” implements most of the components in the sequencer.

To run Vivado Behavioral Simulation, generate memory hex file of the sequencer program by calling program ( ) method with hex\_file argument. In the current Vivado simulator setting, memory file stored at “Sequencer\\v3\_??\\v3\_??.srcs\\sim\_1\\new\\test\_hex.mem” will be loaded into the instruction memory of the sequencer.

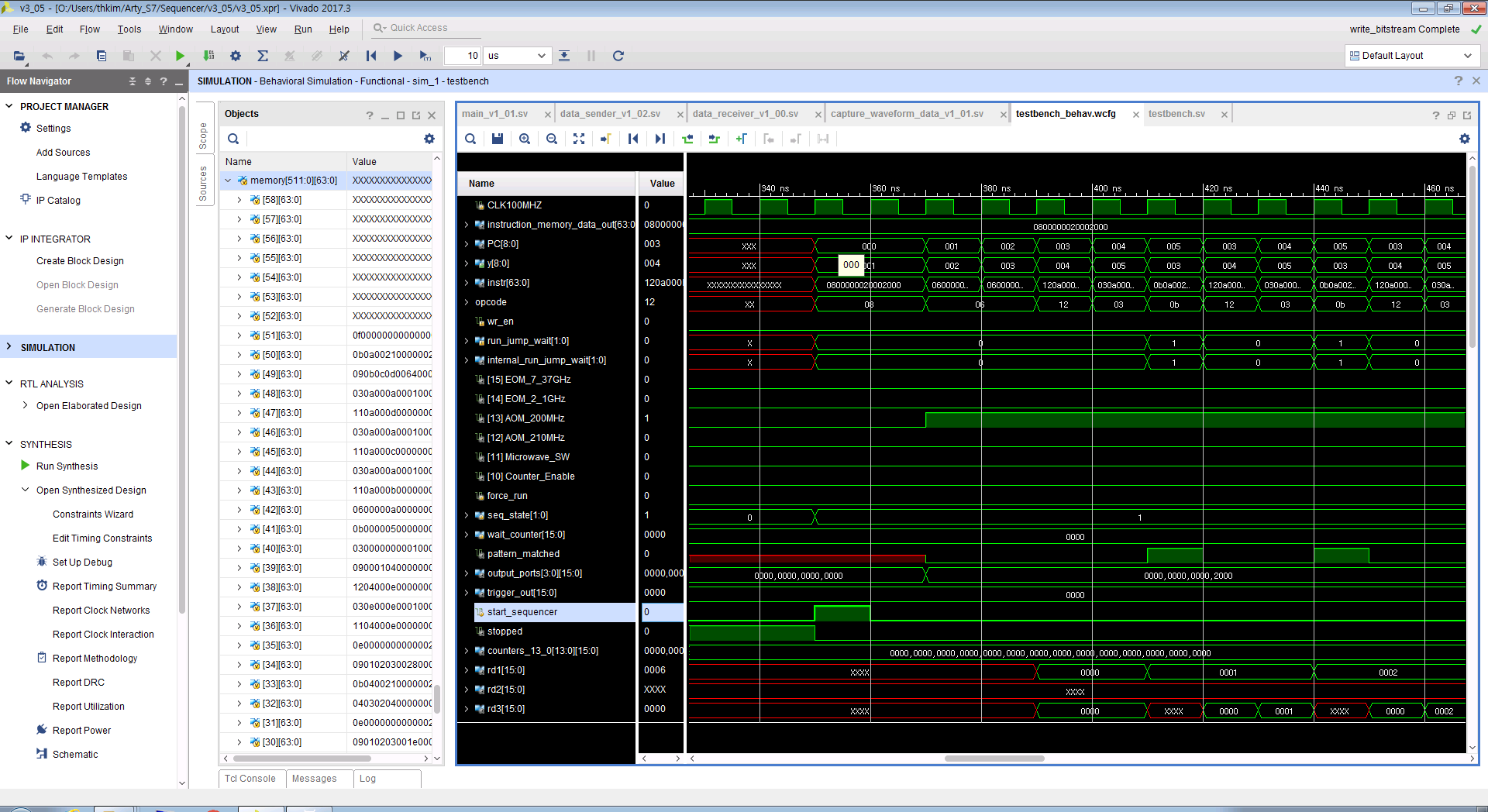
The hex memory file is listed under “Simulation Sources / sim\_1 / Memory File”. After the memory hex file is generated, check the file creation time of the “test\_hex.mem”. Right click on the “SIMULATION” on “Flow Navigator” and choose “Simulation Settings…” Make sure that “Simulation top module name” is testbench. Click “Run Simulation” under “SIMULATION” and choose “Run Behavioral Simulation”.

If there is any problem related to SDB update, right click on “SIMULATION” on “Flow Navigator” again and choose “Reset Behavioral Simulation”.

Once the simulation is started, make sure that the instruction memory is properly loaded with the intended program. To check, click on “instruction\_memory” component under “Scope” tab. Then in the “Objects” window on the right, “memory[511:0][63:0]” item can be found.



Expand “memory[511:0][63:0]” item and scroll down to the lower address and compare the values in the memory with the program. The expected hex value of the program can be found by running program( ) method with machine\_code argument set to True.



In the above screen shot, memory up to address 51 is filled with the program, and memory above address 52 are empty.

The program loaded in the instruction memory will start running from 350 ns because we have to wait until the clock generator will start generating 800MHz clocks.

## Debugging with ILA of Vivado

* Vivado provides the integrated log analyzer (ILA). This will be the most accurate way to probe all the pins, but it takes a lot of time to synthesize a logic with the integrated analyzer, and sometime, it might be possible that it cannot be synthesized due to large logic. There is also a chance that the timing might be different from the normal logic due to extra components.
* <https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug936-vivado-tutorial-programming-debugging.pdf>
* Lab1 Step3:

“Open Synthesized Design”

Click “Netlist” tab in the source tree window

Mark the interesting pins with “Debug”

Click “Set Up Debug” under “SYNTHESIS” pane

“ILA Core Options” page at the end of “Set Up Debug” dialog, choose “Capture control” and “Advanced trigger”.

“Generate Bitstream”

## Debugging with waveform capture

If we uncomment the waveform capture parts in the main Verilog file, we can also monitor the waveform around the specified trigger. It can capture maximum 16-bit pins for 1024 clocks. Similar to ILA, it takes a while to synthesize a total logic.