Iowa State University

R2U2 System Documentation

June 21, 2019

Contents

1 Nomenclature

All the informaiton requried to use R2U2 is called the *configuration*. A R2U2 configuration incudes *design-time configuration* and *runtime configuration*. The design time configuration provides additional informtion to an *implmentation* of the R2U2 engine to create a *unit*. A depoloyed unit can monitor different specificaitons at runtime by loading different runtime configurations.

R2U2 Configuration A combination of the requried design time and runtime infoamtion to deploy R2U2

Design-time Configuration Settings for tailoring the R2U2 engine for a deployment, e.g. compiler settings, FPGA parameters, etc.

Runtime Configuration Specificaitons to be monitored by R2U2

Implmentation The source of an R2U2 engine, written in a programming or hardware description language

Unit The result of configuring an R2U2 implmentation, executable bytecode or sythezized hardware

Deployment One or many R2U2 units targeted at, and configured to provide system health monitoring for, a system.

2 R2U2 Software (C)

2.1 Process Overflow

Figure 1 shows the procedure of running the R2U2 C version by the user. From the high level, the user need to specify the MLTL formulas first. The formula will be converted into 1) binary instruction file; 2) SCQ size assignment file; 3) intervals for each temporal instruction. The user should also prepare the sensor trace file (*.trc) for verification.

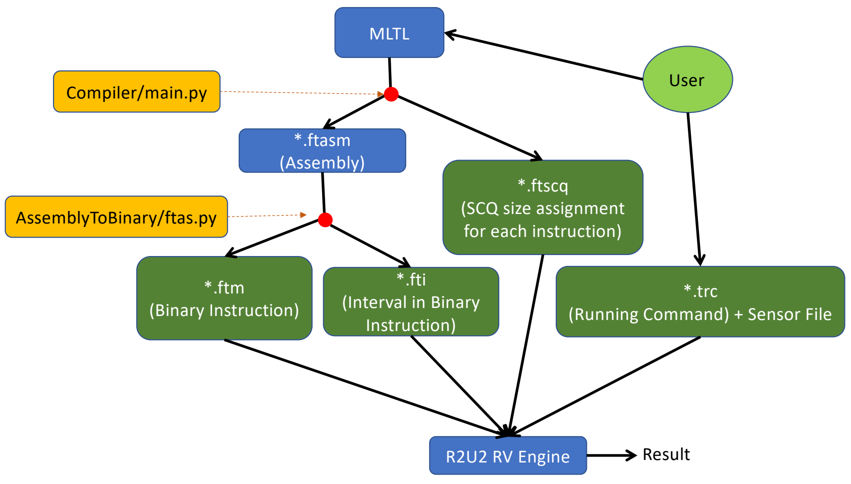

MLTL
User
Compiler/main.py
*.ftasm (Assembly)
*.ftscq (SCQ size assignment for each instruction)
AssemblyToBinary/ftas.py
*.ftm (Binary Instruction)
*.fti (Interval in Binary Instruction)
*.trc (Running Command) + Sensor File
R2U2 RV Engine
Result

Figure 1: Deploying steps of R2U2 C. Blocks marked in green are the files required for the program execution. The yellow blocks tells the dependency of the tool scripts that handle the conversion.

2.2 R2U2 C Operation Modes and Trace Input Format

There are two operation modes for R2U2 C: 1) Offline mode (regression test) 2) Online mode (embedded application).

To select the operation mode, comment/uncomment the line:
"#define ONLINE_MODE" in r2u2/R2U2_SW/R2U2_C/src/R2U2.c
and recompile the C program. Under different modes, the trace input is different (check Table 3).

Table 1: Format and content of trace file

Mode	Offline Mode				Online Mode			
Column Information	sensor 0	sensor 1	sensor 2	...	Command	sensor 0	sensor 1	...
*.trc File Content	1.0	2.0	1.2	...	-1	1.0	2.0	...
	0.8	1.2	1.3	...				
	0.5	1.1	1.2	...				
	...			...				

Under the online mode, the command is included into the *.trc file. The trace file should be updated at each time stamp. In order to specify the behavior for R2U2, we use the first column in the trace file as the command to R2U2. The **command** variable can only be integers. If command=-2, R2U2 will terminate. If command=-1, R2U2 will reset the buffer and prepare for the new round of execution. The positive number of command represents the timestamp of the sensor value. Users should guarantee the timestamp will increase 1 by 1 starting from 0. Table 2 summarizes the meaning of the **command** variable.

Command Value	Description
-2	Terminate the current execution.
-1	Reset the instruction/data etc. buffer to restart the RV. Users must set command to -1 as the RV starts signal or the RV engine will not start.
i (Positive Integers)	Represent the sensor signal at time stamp i. The i should increase by 1 each time, starting from 0. (Increasing i tells the RV engine that new sensor value is coming.)
others	Unspecified

Table 2: Meaning of 'command' variable in *.trc file.

2.3 Atomic Conversion Function

Users need to manually specify the atomic conversion function. The function is specified in file: r2u2/R2U2_SW/R2U2_C/src/AT/at_conversion.c .
Below is an example of the conversion function:

```
void at_checkers_update(const r2u2_input_data_t *r2u2_input_data){
    for (int i=0; i< N_ATOMICS; i++){
        atomics_vector[i] = AT_COMP((r2u2_input_data[i]), > , 0.5);
    }
}
```

,where atomics_vector[i] corresponds to atomic a_i in the MLTL formula. r2u2_input_data[i] corresponds to *sensor i* in the trace file. Uses can borrow some math marcos in at_checkers.h under the same folders or write their own math functions.

2.4 Command Line R2U2 C Execution

Table 3: Software dependency and the prerequisites

MLTL Compiler	version $\geq$ python 3.0
	ply; (pip install ply)
R2U2 C	pkg-config; (sudo apt install pkg-config)
	fftw3; (sudo apt-get install libfftw3-dev libfftw3-doc)

After determining the running mode and the trace file, now we can build the C project and check the execution output. Here is the steps for deploying the R2U2 C:

1. Compile the C program:

   - Navigate to the src folder

     ```
     cd r2u2/R2U2-SW/R2U2_C/src/
     ```

   - Call MakeFile to compile

     ```
     make
     ```

2. Generate configuration files:

   - Navigate to the tools/ folder:

     ```
     cd r2u2/tools/
     ```

   - Convert MLTL formula to assembly:

     ```
     python Compiler/main.py [MLTL formula]
     ```

     This command will generate **tmp.ftasm** and **tmp.ftscq** configuration files. For the syntax of MLTL formula, refer to ??. (Note: special symbol (e.g., &,(,),*space*, etc.) should put a '\' in the front).

   - Convert assembly to binary text:

     ```
     %#cat tmp.ftasm | python AssemblyToBinary/ftas.py tmp
     python AssemblyToBinary/ftas.py tmp.ftasm 4
     ```

     This results the file **tmp.ftm** and **tmp.fti**.

3. Specify the trace file and start

   ```
   bin/r2u2 [trace file *.trc]
   ```

   If under online mode, then the user is in charge of updating the trace file at each time stamp.

3 R2U2 Software (Cpp)

(Require C++11)

3.1 Assembly Mode Configuration

In main file MTL.cpp:

1. Assign the sensor number, simulation time stamps;
2. Link the sensor to the .log file;

```
sensor[0]=new Event("./src/alt.log");
sensor[1]=new Event("./src/pitch.log");
```

In assembly code file test.ftasm:

1. Write the assembly code in test.ftasm
2. Specify the assembly code file

```
string asm_file="./src/test.ftasm";
```

3.2 MLTL Mode Configuration:

In main file MTL.cpp:

1. Assign the sensor number, simulation time stamps;
2. Link the sensor to the .log file;
3. Write your MTL formula as a string in MTL.cpp. For MTL format, see Notes;

3.3 How to Run

In terminal:

1. run setup.sh to compile the project

   ```
   bash setup.sh
   ```

2. type ./Debug/MTL to running MTL

   ```
   bash ./Debug/MTL
   ```

Notes: Meaning the MTL String:

```
Algorithm 1: "!" = "NOT{}"
Algorithm 2: "G[2]" = "KEP[2]{}"
Algorithm 3: "&" = "AND{,}"
Algorithm 4: "G[2,5]" = "ALW[2,5]{}"
Algorithm 5: "F[1,2]" = "UNT[1,2]{,}"
```

```
string formula="NOT{S[0]}";
string formula="NOT{NOT{S[0]}}";
string formula="KEP[5]{S[1]}";
string formula="NOT{KEP[2]{NOT{S[1]}}}";
string formula="AND{KEP[2]{S[0]},S[1]}";
string formula="ALW[5,10]{S[1]}";
string formula="NOT{AND{ALW[5,10]{S[0]},KEP[2]{S[1]}}}";
string formula="AND{KEP[2]{NOT{NOT{S[0]}}},S[0]}";
string formula="AND{S[1],ALW[0,8]{S[0]}}";
string formula="AND{S[1],KEP[8]{S[0]}}";
string formula="AND{ALW[5,10]{S[0]},KEP[2]{S[1]}}";
string formula="KEP[2]{S[1]}";
string formula="AND{AND{S[0],S[1]},ALW[3,5]{S[0]}}";
```

4 R2U2 Software (Python)(Requires python 3.X)

The current python version only supports offline test. Python R2U2 is suitable for research purpose if you want to check how the SCQ works or the observer algorithm. The python version code is located under the repository: `r2u2/R2U2_SW/R2U2_PYTHON/` .

4.1 Process Overflow

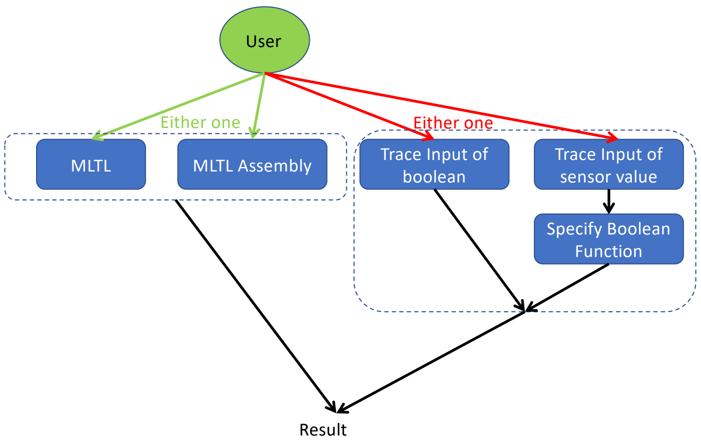


Figure 2: Deploying steps of R2U2 Python.

User can run the R2U2 python in terminal:

```
   python MLTL_main.py -m "$(cat [A])" -s [B] -o [C]
```

A: Assembly/MLTL Input File: Assembly code in a file or MLTL formula written in a file
B: Signal Trace File. Refer to Section 4.2 for detail format.
C: Output File. The RV result will be written into the file. If user did not specify **-o** option, the output file will be default 'untitled.txt'.

Or

```
   python MLTL_main.py -m [A] -s [B] -o [C]
```

A: MLTL formula (Take care of special symbol in command line)
B: Signal Trace File. Refer to Section 4.2 for detail format.
C: Output File. The RV result will be written into the file. If user did not specify **-o** option, the output file will be default 'untitled.txt'.

4.2 Input Trace Format and Boolean Function

Table 4 shows the input format for the trace input file. In the trace input file, the first row is the

Table 4: Format and content of trace file

Mode	Atomic Input Format(0/1)				Sensor Input Format(floating)			
Column Information	atomic 0	atomic 1	atomic 2	...	sensor 0	sensor 1	sensor 2	...
trace file content	a0	a1	a2	...	altitude	velocity	temp	...
	0	1	1	...	1005	32.7	87	
	0	0	1	...	1010	-1.7	87	
	...			...	...			

trace name. After the first row, each line is the data in a seperate time stamp ranging from 0. E.g., the second row is data in time stamp 0, third row is data in time stamp 1. For each column (split by either space or comma), the first column data is signal trace 0. The second column data is signal trace 1 and so on.
Note that 1) For atomic input format, the atomic name should be consistent with the atomic name in MLTL specification; 2) For sensor Input, the sensor name should be consistent with the boolean function in r2u2/R2U2_SW/R2U2_PYTHON/ACOW/Traverse.py function s2a(self,signal_trace). The atomic map in the function should be consistent with the MLTL specification. Below is the example for function s2a(self,signal_trace):

```
def s2a(self,signal_trace):
    atomic_map = {}
    s2d = {signal_name:i for i,signal_name in enumerate(self.trace_name)}

    ####################################################################
    # For User: map boolean function to atomic
    atomic_map['a0'] = abs(s2d['altitude'])<0.04
    atomic_map['a1'] = abs(s2d['velocity'])<0.08
    atomic_map['a2'] = s2d['temp']>0.6
    ####################################################################
    return atomic_map
```

4.3 AST Optimization

AST optimization only works in MLTL formula input. If you want to optimize the AST, change the input argument in function formula_input(). The following line of code in MTL_main.py:

```
    valid_node_set = formula_input(MLTL) # optimize
    valid_node_set = formula_input(MLTL,False) # unoptimize
```

4.4 Model Checking (not used in R2U2)

4.4.1 Define Automata

You can specify the input as the automata.

```
def define_automaton():
    a = automaton()
    #sg = StateGen(3,'s','a') # call function to autogenerate state for you
    a.INITIAL_STATE = 's2'
    a.DEST_STATE = 's3'
    #a.STATE, a.DELTA = sg.gen_aut()
    a.STATE = {
    's0':{'a0':0,'a1':0},
    's1':{'a0':0,'a1':1},
    's2':{'a0':1,'a1':0},
    's3':{'a0':1,'a1':1},
    }
    a.DELTA = {
    's0': {'s0','s1','s2','s3'},
    's1': {'s0','s1','s2','s3'},
    's2': {'s0','s1','s2','s3'},
    's3': {'s0','s1','s2','s3'},
    }
    print(a)
    return a
```

4.4.2 Run Model Checking

The tool supports model checking for a MLTL formula given the automata. User need to define the automata with initial state **INITIAL_STATE** and desired end state **DEST_STATE**.

```
    a = define_automaton()
    a.init()
    valid_node_set = assembly_input(MLTL)
    solution = Search(a,valid_node_set,agent='DES')
```

5 Run R2U2 in Vivado Simulation

(Tested in Vivado 2017.02)

5.1 Name of Binary Files:

The HW simulator takes the binary assembly code as input. There are three binary files that are necessary to run the HW simulation:

- *.trc: Input signal at each time stamp;
- *int: Intervals of certain operations;
- *imem: Binary assembly code without interval.

5.2 Steps to build the project:

1. In vivado command window, **cd** into the folder **/r2u2/R2U2_HW/Hardware/hdl/ftMuMonitor/vivado**
2. Click Tools→ Run Tcl Script. Then Choose the .tcl file under current directory
3. Rewrite the *.trc, *.int, *.imem to test different cases (Refer to "How to generate binary file for the HW test")
4. The async result is shown in the file **async_out.txt** under the folder **/r2u2/R2U2_HW/Hardware/hdl/ftMuMonitor/vivado/FT_Monitor/FT_Monitor.sim/sim_1/behav**

5.3 How to generate binary file for the HW test:

1. Write the assembly code in casestudy.ftasm under directory **r2u2/R2U2_HW/Software/ftAssembler/**
2. run **./convert.sh** under that folder. The script will call the python script to generate the binary file (*.imem, *.int). It will automatically copy these files into the vivado project.
3. The input signal is called **atomic.trc** located under **r2u2/R2U2_HW/Hardware/hdl/ftMuMonitor/sim/testbench/**. Each signal is written into a column. The first signal is in column 0.

6 Running on the FPGA

This section talks about how to generate required files and steps to run the demo.
Note: 1) The text mark with 🟦 means the command typed in terminal.
2) The text marked in green are the files that require manually modify.

6.1 Process Overflow

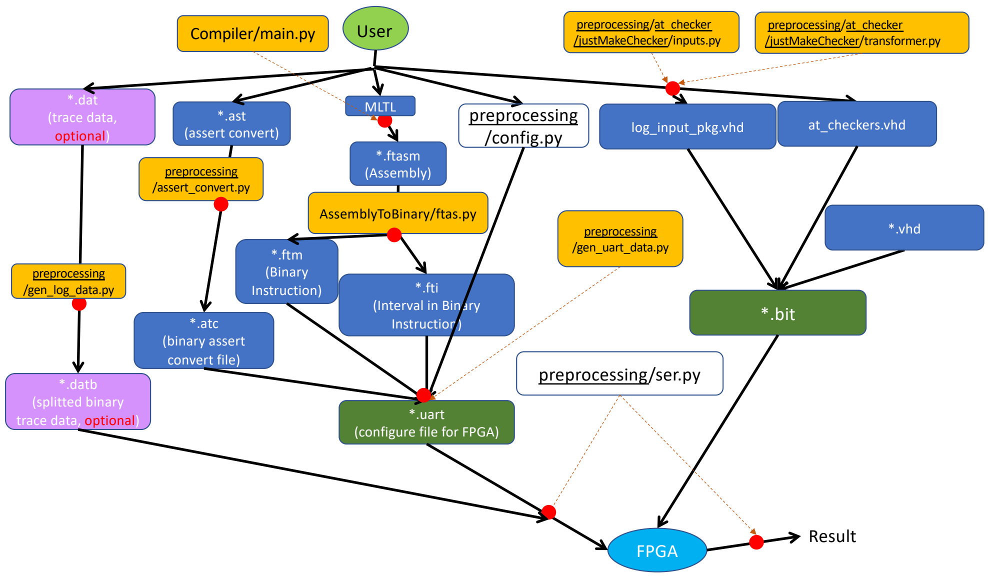

Compiler/main.py
User
preprocessing/at_checker/justMakeChecker/inputs.py
preprocessing/at_checker/justMakeChecker/transformer.py
*.dat (trace data, optional)
*.ast (assert convert)
MLTL
preprocessing/config.py
log_input_pkg.vhd
at_checkers.vhd
preprocessing/assert_convert.py
*.ftasm (Assembly)
AssemblyToBinary/ftas.py
preprocessing/gen_uart_data.py
*.vhd
preprocessing/gen_log_data.py
*.ftm (Binary Instruction)
*.fti (Interval in Binary Instruction)
*.bit
*.atc (binary assert convert file)
preprocessing/ser.py
*.datb (splitted binary trace data, optional)
*.uart (configure file for FPGA)
FPGA
Result

Figure 3: Deploying steps of R2U2 into an FPGA.

6.2 Introduction to Configuration File

To generate the binary configuration file for the FPGA, user need to provide the following dependent files:

1. **atomic checker (VHD file)**: All the sensor signals should be processed into atomic propositions in R2U2 by defining the atomic conversion functions. These functions will be conducted in hardware. User need to specify how many atomic checkers are required in hardware and the sensor input into each atomic checker. This procedure results in two VHDL files: at_checkers.vhd and log_input_pkg.vhd. We have tools to support building the VHDL file, namely inputs.py, transformer.py and util.py. User only need to modify inputs.py. For details of inputs.py, refer to B.1.

2. **assert conversion configuration file**: Apart from specifying the atomic checker for the hardware, users also need to use the software to configure the atomic conversion function during R2U2 runtime. Basically, the configuration tells what number you want the sensor signal to compare with, or do you want to do the addtion/minus of two sensor signals first before the comparasion. The rules is written in the file *.ast. For the syntax of the rules, refer

to B.2. Script assert_convert.py will convert the rules into binary format that the hardware can decode.

3. **MLTL related file**:
4. **sensor trace file**: User manually type the input in binary format as the sensor signal at each time stamp.

User should modify the config.py file first before running any scripts. In the config.py file, user need to specify the following parameters:

6.2.1 Run Mode

We can specify the running mode in config.py file. The monitor has two running mode:

1. **self_sensing**: Read sensor signal on chip under the specified sample rate. This is what we show in the Oct.2018 demo video.
2. **read_log**: Read sensor signal from UART software. In this mode, the processed sensor in raw binary format will be send to FPGA through UART. The purpose of this mode is for debugging or regression test.
3. **type_input**: User manually type the input in binary format as the sensor signal at each time stamp.

6.3 Steps-Preparation

Navigate to folder /tools/preprocessing/

1. Modify general configuration file
   First, modify the configuration file config.py. This file sets the data width and running mode for R2U2.
2. Generate atomic checker associated file
   Go to folder /tools/preprocessing/at_checker/
   (a) Modify inputs.py (section B.1) to specify the input data's format and name.
   (b) Generate at_checkers.vhd and log_input_pkg.vhd
   ```
   python transformer.py
   ```
   (c) Write atomic assertion configuration in input.ast (section B.2). Go to folder /tools/preprocessing/, use the following command to convert the assertion into binary configuration file named res.atc.
   ```
   python assert_convert.py
   ```
3. Generate binary instruction assembly code and its interval file (.ftasm, .ftm and .fti)
   (a) In folder tools/preprocessing/
   ```
   (python 3) python ../Compiler/MLTL_main.py [MLTL]
   ```

This command generates assembly file tmp.ftasm from MLTL.

```
cat tmp.ftasm | ../AssemblyToBinary/ftas.py tmp
```

This command takes tmp.ftasm as input and generates binary instruction file tmp.ftm and tmp.fti.

4. Generate UART byte data Run following command to generate *.uart file. This file will be send through uart to FPGA.

```
python gen_uart_data.py
```

Note for VHDL code:

1. Current VHDL R2U2 uses the fixed SCQ size for each MLTL assmebly instruction. User can change the SCQ size by modifying parameter QUEUE_SIZE in ft_mu_monitor_pkg.vhd.

2. To change the timestamp width, 1) revise parameter TIMESTAMP_WIDTH in mu_monitor_pkg.vhd. 2) Revise TIMESTAMP_BYTE_extend_byte in config.py (unit in byte).

6.4 Steps-Execution

Navigate to folder /tools/preprocessing/, start the uart communication with FPGA.

```
(python 2) python ser.py
```

This command will send *.uart to FPGA first. Based on the running mode in config.py, the script will fall into one of the following operation:

1. **self_sensing mode**: Nothing need to be done. The script will periodically output the verification result. User can use the following command to reset R2U2:

```
(python 2) python reset.py
```

2. **read_log mode**: Nothing need to be done. The script will automatically read the *.datb log data to FPGA and display the result back on the terminal. Once finished verifying all the log data, R2U2 will self reset.

3. **type_input mode**: User should type binary data as input (same format as each line in logged_data.dat) and press enter. You will see the result displayed in the terminal window.

6.4.1

6.4.2 Run on zynq test board

Hardware Connection:

- FPGA Board Connection
  1. Connect the JTAG with PC for downloading bitfile. See Figure 5.
- UART Connection Connect the UART module with PC, connect UART module PIN **DIN** with Zedboard **JP4**, PIN **DOUT** with Zedboard **JP3**. See Figure 6.
- Reset button is located in Zedboard **BTNC(P16)**.

Figure 4: Connect zybo with USB module

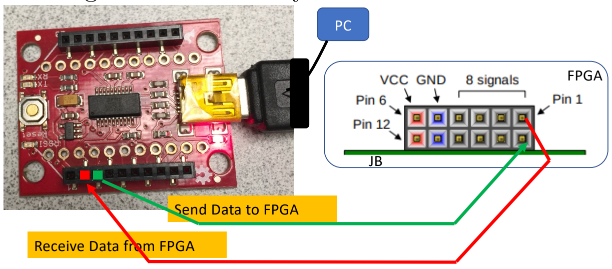


Figure 5: Zedboard connection setup

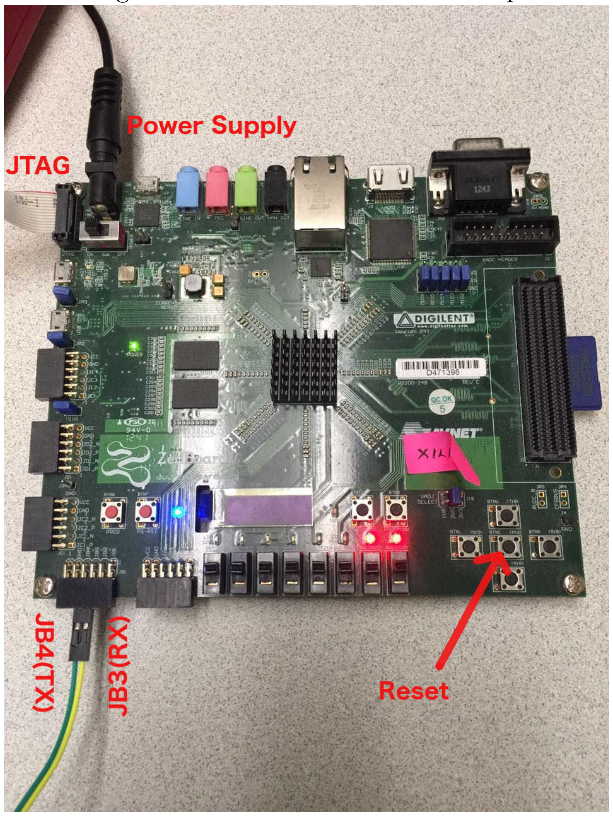
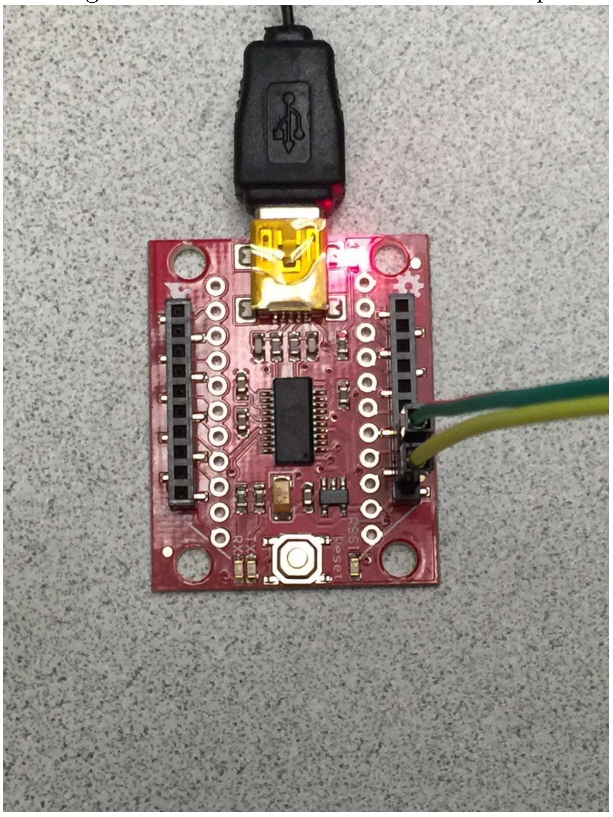


Figure 6: UART module connection setup



A File Structure

```
scripts/
├─config.py (setup all the configuration parameters in this file)
│
├─at_checker/(generate VHDL file for atomic checker)
│  └─inputs.py(list all the atomic checkers and signals width here)
│
├─ftAssembler/
│  ├─casestudy.txt (write the MTL formula here)
│  │
│  ├─rawConvert.sh (script to compile MTL into assemply)
│  │
│  └─compiler/
│
├─sensor_data.dat (write the sensor data here for simulation purpose)
│
└─gen_log_data.py (convert the sensor_data.dat into binary format)
```

B Details of Scripts and Files

B.1 inputs.py

(only compatiable with python 2.x)
Processing raw data file (similar to .csv) to the data format we want.
Parameters and functions:

1. DATA_SET: Raw data set file
2. class subclass(CsvParser):
   e.g.

```
class Gs111m(CsvParser):
    def __init__(self):
        CsvParser.__init__(self)
        self.file = DATA_SET + "/gs111m_0.csv"
        self.addConfig(9, "float", 10, 24, "roll_angle", "in 1/2^10 rad")
        self.addConfig(8, "float", 10, 24, "pitch_angle", "in 1/2^10 rad")
```

   Comment: Create a subclass. It will subtract the data you mentioned in **self.addConfig** and output the processed data into **self.file** The self.file will look like:

roll_angle	pitch_angle
-42	12
-34	6
-25	3
...	...

▷ self.addConfig(channel, type, comma, width, name, comment)
@ channel: Column of the data in the raw data file.
@ type: Float or not. If it's float, write "float", else leave null.
@ comma: Only used for floating data type. It specifies how many fraction bit in binary you want to reserve during the conversion.
@ width: Number of bits for this data.
@ name: Specify the name of the column data. This will affect the name used in the .vhd code.
@ comment: Add any comment in string as you want.

3. class subclass(AtChecker):
   e.g.

```
class AtCheckerConfig(AtChecker):
    def __init__(self, inputFiles):
        AtChecker.__init__(self, inputFiles)
        self.add("pitch_angle", "", 1, "", "", "", "")
        self.add("roll_angle", "", 1, "", "", "", "")
```

   Comment: Create a subclass. The subclass specifies the filters, number of atomic checkers, etc.. The class will affect at_checker.vhd
   ▷ self.add(input1, input2, count, filter1, filter2, rate1, rate2)
   @ input1: First input to the at_checker
   @ input2: Second input to the at_checker, usually used for compare with @input1. We can leave it as a null string "".
   @ count: How many atomic checkers you want for this signal.
   @ filter1: Hardware filter you want to use for input1. The filter name should be the same as the hardware filter component
   @ filter2: Hardware filter you want to use for input2.
   @ rate1: Signal delta during each sampling clk for input1. Leave null if you want to monitor signal delta.
   @ rate2: Signal delta during each sampling clk for input2.

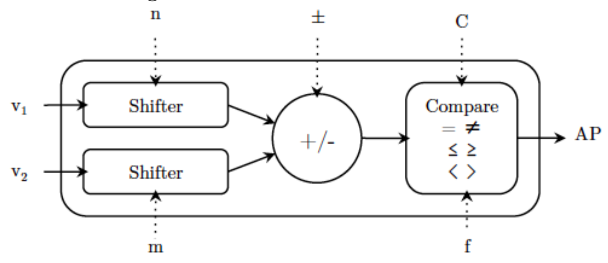
B.2 input.ast

Atomic assertion file. Specify the at_checker.vhd operation mode. For details, refer to **Performance_Aware_Hardware_Runtime_Monitors.pdf**.
e.g.

```
-16;0;0;+;>
1;0;0;+;=
100;0;0;+;>
```

Each line specifies the configuration for one atomic checker block. Consider the structure of an atomic checker, depicted in Figure.7, then each line in the configuration file reads as: C;n;m;+/-;f.

Figure 7: Atomic checker block



B.3 gen_uart_data.py

Generate uart data byte by byte. Requires **.atc**, **.imem**, **.int** and **.dat** as input. I suggest leave the **.dat** file empty for the demo purpose.
Parameters:
@ SETUP_DATA_WIDTH_extend_byte: **.atc** file configuration data width.
@ SETUP_ADDR_WIDTH_extend_byte: **.atc** file configuration address width.
@ DATA_BYTE_WIDTH_extend_byte: binary logged data bit width (each line width of logged_data.dat).
These three parameters should be the same as in R2U2_pkg.vhd.

B.4 ser.py

1. serial.Serial()
   e.g.

   ```
   ser = serial.Serial(
       port='/dev/ttyUSB0',
       timeout=0,
       # baudrate=9600,
       # parity=serial.PARITY_ODD,
       # stopbits=serial.STOPBITS_TWO,
       # bytesize=serial.SEVENBITS
   )
   ```

   Comment: By default, it is 9600 baud rate 8IN1 mode. You only need to specify the PC port that the UART is connecting to.

2. parameters: @ DATA_BYTE_WIDTH_extend_byte: (same as gen_uart_data.py)