

Auto-Assessment Platform (AASP)	2
Judge0	3
Waveform Visualization	9
AASP Integration	18

Auto-Assessment Platform (AASP)

Introduction

Welcome to the AASP developer documentation! This documentation aims to explain the code implemented during my [Final Year Project](#) (FYP). My main contribution was adding Hardware Description Language (HDL) support to AASP. Code implemented by previous FYP, including the [original AASP](#) and [improved version](#) with test proctoring, will not be covered.

A demo video covering the main features of my contributions can be found [here](#).

Source Code

The central repository for AASP can be found [here](#), where all code references in this documentation can be found. The repository is further split into the following sub-repositories:

Repository	Description
aasp-core	Core of AASP, containing the web application.
aasp-judge0	Forked from judge0 , a free and open-source online code execution system. Modified to suit the needs of AASP.
aasp-compilers	Forked from judge0-compilers , a collection of compilers used by Judge0. Modified to include HDL compilers and additional parameters.
aasp-vcd2wavedrom	Forked from vcd2wavedrom , a tool to convert VCD files to WaveDrom format.

Content

The following implementations will be covered in this documentation:

Implementation	Description
Judge0	HDL is not a supported language by Judge0, the code execution engine used by AASP. The Judge0 core and compiler image are modified to support HDL for AASP.
Waveform Visualization	WaveDrom and VCDrom are used to generate static and interactive waveforms for AASP. Additional features are added to the tools to fit AASP's use case.
AASP Integration	Integration of the above implementations into AASP.

Contact Me

If you have any questions regarding this work, feel free to reach out to [me](#).

Judge0

Overview

[Judge0](#) is an open source code execution engine used by AASP for code compilation and execution. However, HDL is not one of its supported language. This section will cover the steps performed to add Verilog HDL to Judge0. Adding other programming languages can follow a similar process outlined in this section, with a few modifications based on the specifications of the desired language to be added.

This section will cover the following:

- [Overview](#)
- [Source Code](#)
- [Modifications](#)
 - [Adding Verilog HDL](#)
 - [Adding Additional Attributes](#)
- [API Usage](#)
- [Supported Languages](#)

Source Code

[Judge0](#) and [Judge0 Compilers](#) are the two repositories AASP uses. They are forked from the original repositories, and changes were made to support Verilog HDL.

Modifications

Two modifications were implemented to Judge0 for AASP:

- Adding Verilog HDL
- Adding additional attributes

 Modifying Judge0 will require basic Ruby knowledge since the Judge0 codebase is written in Ruby.

Adding Verilog HDL

To add a new programming language to Judge0, we must first find the necessary compiler/interpreter for that particular language. In this case, [Icarus Verilog \(iVerilog\)](#) is chosen as the compiler for Verilog HDL. Once that has been established, we must find a [download link](#) for the said compiler.

Since all compilers are located in the Judge0 compiler image, iVerilog needs to be installed there. The code below shows a snippet of the compiler Dockerfile that will install iVerilog. The full Dockerfile for AASP can be found [here](#).


```
1 # Check for latest version here: https://github.com/steveicarus/iverilog/releases
2 ENV IVERILOG_VERSIONS \
3     v11-branch
4 RUN apt-get -y update && \
5     apt-get install -y \
6     automake \
7     autoconf \
8     gperf \
9     build-essential \
10    flex \
11    bison \
12    git && \
```

```

13 rm -rf /var/lib/apt/lists/*
14 RUN git clone --branch=${IVERILOG_VERSIONS} https://github.com/steveicarus/iverilog && \
15     cd iverilog && \
16     bash autoconf.sh && \
17     ./configure && \
18     make && \
19     make install && \
20     cd && \
21     rm -rf iverilog

```

Once you have the updated Dockerfile, you can proceed to build the image. You should also publish the image to [Docker Hub](#), and [update](#) the main Judge0 Dockerfile to use this new image.

 Building the compiler image will take a long time, especially if you're installing more compilers.

Over on the main Judge0 image, we will need to update the [active language list](#) to inform Judge0 of this new update.

```

1 @languages ||= []
2 @languages +=
3 [
4   {
5     id: 69,
6     name: "Plain Text",
7     is_archived: false,
8     source_file: "text.txt",
9     run_cmd: "/bin/cat text.txt"
10  },
11  ...
12  {
13    id: 89,
14    name: "Multi-file program",
15    is_archived: false,
16  },
17  {
18    id: 90,
19    name: "Verilog (Icarus Verilog 11.0)",
20    is_archived: false,
21  }
22 ]

```

For single file programs, where only 1 source file is needed to compile your program, the metadata such as the source file name, compile and run command will need to be specified here. However, since Verilog will be a multi-file program, we will notify Judge0 during submission. This will be explained in more details in the API usage section.

 Note that you **should not** update the ID of the languages unnecessarily, since this can change the compiler used by AASP.

Since Verilog will be a multi-file program, we need to indicate it as such. Judge0 uses the `is_project` method in the [language model](#) to identify, by verifying the name of the language.

```

1 class Language < ApplicationRecord
2   validates :name, presence: true
3   validates :source_file, :run_cmd, presence: true, unless: -> { is_project }
4   default_scope { where(is_archived: false).order(name: :asc) }
5
6   def is_project
7     name == "Multi-file program" || name == "Verilog (Icarus Verilog 11.0)"
8   end
9 end

```

Adding Additional Attributes

AASP's frontend requires the Value Change Dump (VCD) file outputted by the compilation process to generate the interactive waveform using VCDrom. We return this VCD output from Judge0 by adding an additional [submission attribute](#).

The list of submission attributes are defined in the submission model [file](#), and an additional parameter `vcd_output` is added to return the VCD output from the compilation process.

```
1 class Submission < ApplicationRecord
2   ...
3   def vcd_output
4     @decoded_vcd_output ||= Base64Service.decode(self[:vcd_output])
5   end
6
7   def vcd_output=(value)
8     super(value)
9     self[:vcd_output] = Base64Service.encode(self[:vcd_output])
10  end
```

Update the [submission serializer](#) as well.

```
1 class SubmissionSerializer < ActiveModel::Serializer
2   attributes((Submission.column_names + ["status", "language"] - ["id"]).collect(&:to_sym))
3
4   def self.default_fields
5     @@default_fields ||= [
6       :token,
7       :time,
8       :memory,
9       :stdout,
10      :stderr,
11      :compile_output,
12      :vcd_output,
13      :message,
14      :status
15    ]
16  end
17  ...
18  def vcd_output
19    object_decoder(:vcd_output)
20  end
```

Next, we need to update the isolate job, the script that creates the sandbox for each submission, to add the new attribute to our submission object. The file name of the VCD output must be known in order to be read. The file name will be defined during our submission, which we will fix to `vcd_dump.vcd`.


In particular, Verilog specifies the VCD dump configurations in the testbench:

```
1 initial begin
2   // nititalize VCD dump
3   $dumpfile("vcd_dump.vcd");
4   $dumpvars(0);
5 end
```

The `$dumpfile` specifies the file to dump the changes in, and `$dumpvars` specifies which variables are to be dumped. For AASP, we want to specify that all variables will be dumped to the `vcd_dump.vcd` file. To ensure that all testbenches sent for compilation has this declaration,

we will inject them during the code compilation process.

```
1 if testbench.find('$dumpfile') == -1:
2     # add wave dump to last line before endmodule
3     testbench = testbench.replace('endmodule', 'initial begin $dumpfile("vcd_dump.vcd"); $dumpvars(0); end endmodule')
4 else:
5     # Define the regular expression patterns
6     dumpfile_pattern = r'\$dumpfile\[^\]+\\"
7     dumpvars_pattern = r'\$dumpvars\(\d+\)'
8
9     # Replacement strings
10    new_dumpfile = '$dumpfile("vcd_dump.vcd")'
11    new_dumpvars = '$dumpvars(0)'
12
13    # replace wave dump
14    testbench = re.sub(dumpfile_pattern, new_dumpfile, testbench)
15    testbench = re.sub(dumpvars_pattern, new_dumpvars, testbench)
```

 We could specify the modules to be dumped directly in the \$dumpvars declaration, removing the need to modify vcd2wavedrom, but it does not seem feasible since we are not fixing the module declaration.

All we have to do now is to read this file and append it to our submission object.

```
1 vcd_output_file = boxdir + "/" + "vcd_dump.vcd"
2 initialize_file(vcd_output_file)
3
4 # return vcd dump if submission is Verilog (Icarus Verilog 11.0)
5 if submission.language.name == "Verilog (Icarus Verilog 11.0)":
6     vcd_dump = File.read(vcd_output_file)
7     vcd_dump = nil if vcd_dump.empty?
8     submission.vcd_output = vcd_dump
9 end
```

API Usage

Multi-file programs work slightly differently from single-file programs, in that all your metadata and all files should be sent through the `additional_files` attribute, in the form of a Base64 encoded .zip archive. The metadata should include the compile and run scripts for Judge0 to execute. An example of the zip file to be sent through the `additional_files` attribute is shown below:

```
1 # create zip file
2 with zipfile.ZipFile('submission.zip', 'w') as zip_file:
3     zip_file.writestr('main.v', main)
4     zip_file.writestr('testbench.v', testbench)
5     zip_file.writestr('compile', 'iverilog -o a.out main.v testbench.v')
6     zip_file.writestr('run', "vvp -n a.out | find -name '*.vcd' -exec python3 -m vcd2wavedrom.vcd2wavedrom --aasp")
7
8 # encode zip file
9 with open('submission.zip', 'rb') as f:
10    encoded = base64.b64encode(f.read()).decode('utf-8')
11
12 # append the encoded data to your submission params
```

You can then POST the submission to the Judge0 API as shown below:

```
1 params = {
2     "additional_files": encoded,
```

```

3  "language_id": 90, # iVerilog
4  "stdin": test_case.stdin,
5  "expected_output": test_case.stdout,
6  }
7
8  # call judge0
9  try:
10     url = settings.JUDGE0_URL + "/submissions/?base64_encoded=false&wait=false"
11     res = requests.post(url, json=params)
12     data = res.json()

```

Supported Languages

A list of languages relevant to NTU SCSE's curriculum has been curated for AASP's compiler image. Although AASP currently only allows for C, Java, Python and Verilog to be selected, adding any of the supported language will be a trivial task. The list of supported languages is as follows:

#	Name
1	Assembly (NASM 2.14.02)
2	Bash (5.0.0)
3	C (Clang 7.0.1)
4	C (GCC 7.4.0)
5	C (GCC 8.3.0)
6	C (GCC 9.2.0)
7	C++ (Clang 7.0.1)
8	C++ (GCC 7.4.0)
9	C++ (GCC 8.3.0)
10	C++ (GCC 9.2.0)
11	Executable
12	Java (OpenJDK 13.0.1)
13	JavaScript (Node.js 12.14.0)
14	Objective-C (Clang 7.0.1)
15	PHP (7.4.1)
16	Plain Text
17	Python (2.7.17)
18	Python (3.8.1)
19	Ruby (2.7.0)
20	SQL (SQLite 3.27.2)

21	TypeScript (3.7.4)
23	Verilog (iVerilog 11.0.0)*

*: Multi-file language

Waveform Visualization

Overview

Waveform visualization is a core feature added to AASP when integrating HDL support since the outputs generated by HDLs should be visualized intuitively through timing diagrams, rather than using simple text outputs. The waveform visualizations can be categorized into two forms:

- Static waveform visualization
- Interactive waveform visualization

Static waveforms are used to visualize the output compiled, as well as to perform comparison between different outputs. Interactive waveforms allows for a more detailed exploration of the output compiled.

This section will cover the following:

- [Overview](#)
- [Source Code](#)
- [Static Waveform](#)
 - [vcd2wavedrom](#)
 - [Waveform Comparison](#)
- [Interactive Waveform](#)
 - [Time Step Values](#)
 - [Control Buttons](#)
- [Build](#)

Source Code

[VCDrom](#) and [vcd2wavedrom](#) are the two main repositories for the waveform implementations. Modifications were made to fit AASP's use case.

Static Waveform

[WaveDrom](#) is the library used to generate static waveform from simple textual description. An example of a simple waveform generated by WaveDrom is shown below:


Lets start with a quick example. Following code will create 1-bit signal named **"Alfa"** that changes its state over time.

```
1 | { signal: [{ name: "Alfa", wave: "01.zx=ud.23.456789" }] }
```

Every character in the **"wave"** string represents a single time period. Symbol **"."** extends previous state for one more period. Here is how it looks:



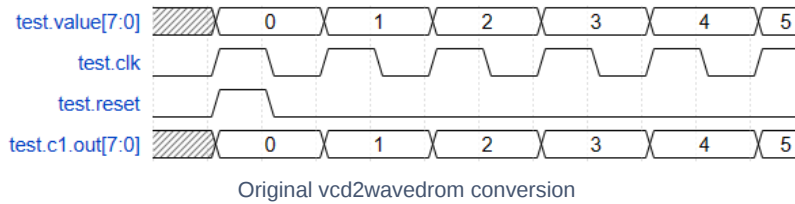
WaveDrom textual description

 To learn more about the syntax of WaveDrom, refer to the [tutorial](#). A simple understanding will be required to make comparisons between different waveforms for AASP.

vcd2wavedrom

To use WaveDrom, we will first need to convert our VCD data from the compilation process to WaveDrom format. The python script for this conversion is taken from [this repository](#), with a few [minor changes](#). The main issue faced when using this script was that all signals were converted, including output signals from both the testbench outputs and the instantiated module outputs. This will clutter the waveform

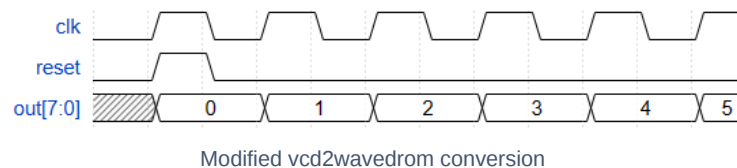
generated with unnecessarily repeated signals. Furthermore, the signals are appended with unnecessary module names. An example is shown below:



A minor modification is done to extract the last reference level and the last signal name:

```

1  aasp = self.config['aasp']
2  if aasp:
3      level = -1
4  else:
5      level = 0
6
7  for i in vcd:
8      if i != '$end':
9          if aasp:
10             if vcd[i].references[level].count('.') > 1:
11                 signal_name = vcd[i].references[level].split('.')[-1]
12             else:
13                 continue
14         else:
15             signal_name = vcd[i].references[level]
```



This conversion has to be done by Judge0 during the compilation step, since we are using the WaveDrom format as the expected output solution for comparison. Below is the [snippet](#) of the updated Dockerfile for our compiler image.

```

1  # Check for latest version here: https://github.com/chongyih/vcd2wavedrom
2  RUN git clone https://github.com/chongyih/vcd2wavedrom && \
3      cd vcd2wavedrom && \
4      pip3 install . && \
5      cd .. && \
6      rm -rf vcd2wavedrom
```

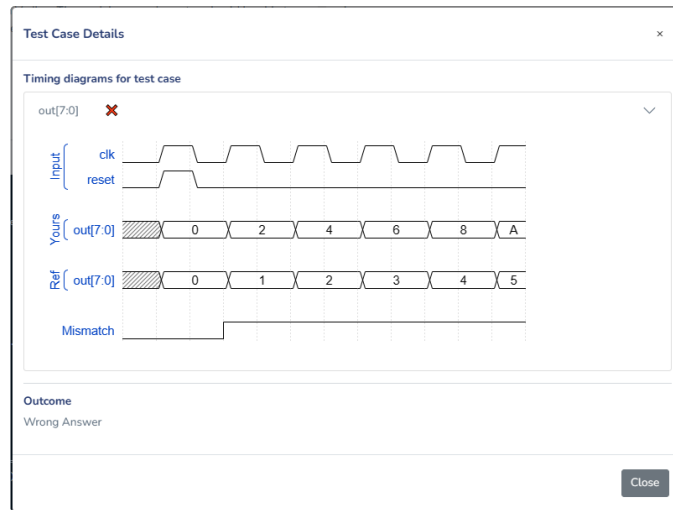
By installing the script into our compiler image, we can execute this script directly. The command below is the run command passed to Judge0 for Verilog compilation as described in the previous chapter.

```

1  vvp -n a.out | find -name '*.vcd' -exec python3 -m vcd2wavedrom.vcd2wavedrom --aasp -i {} + | tr -d '[:space:]'
```

Waveform Comparison

The waveform outputs produced by the student compiled solution and expected solution is compared and displayed in the test case detail modal. The image below shows the comparison WaveDrom graph in AASP.



Waveform comparison graph

The input signals are grouped together, and each output signals are placed side by side, along with a mismatch graph to compare the two output signals. To do so, we will need to be able to identify which signals are inputs and outputs. There is no direct way to achieve this from the VCD file, so it is indicated with an `in_` and `out_` appended in front of the signal name by the frontend before sending the code for compilation.

Once that is established, we can compare the waveforms by grouping the signals according to the port direction. To generate the mismatch graph, we first convert all '.' signals to the previous value to simplify the comparison process. We then loop through the output signal, and compare the character of each string with the other output signal, setting a high if there is a mismatch, and low if there is a match.

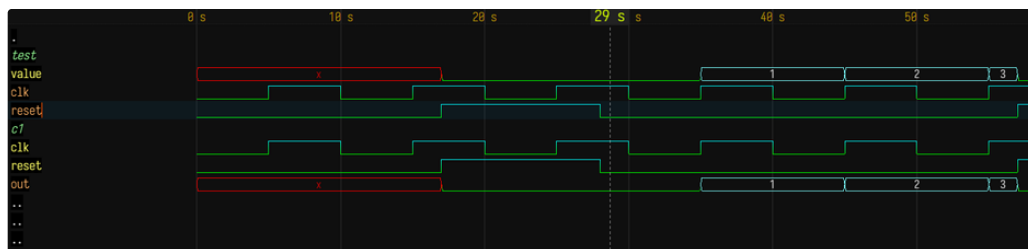
Interactive Waveform

VCDrom is an open source VCD viewer PWA application built on top of various WaveDrom tools. However, VCDrom lacks certain features that are necessary for AASP's use case. The following modifications were implemented to VCDrom:

- Display time step values
- Interactive control button

Time Step Values

VCDrom cannot identify the values of signals at certain time steps. Let's say you want to know what values the signals have at time 29s, you would need to manually trace the signals to identify the current value. If we have a more complex circuit, with multiple signals lasting a long period of time, this can become rather tedious. An example of this problem in VCDrom is shown below:



VCDrom is built on top of [doppler](#), the component that renders the HDL waveforms. It contains a custom domContainer that tracks various states, such as xOffset and yOffset. Furthermore, it contains the elements and layers responsible for displaying everything, from the main container, to the grids and cursor. It also accepts plugins that will continuously be called each time the user moves their mouse.

To add time step values to VCDrom, we will first need to add a corresponding element and layers to display these values.

```
1 const elemento = mountTree.defaultElemento
2 elemento['time'] = ['div', {class: 'wd-time'}]
3 elemento['selectvalue'] = ['div', {class: 'wd-selectvalue'}]
4
5 const layers = mountTree.defaultLayers.concat('time').concat('selectvalue');
```

The time layer will render a line at the selected time step even after the user moves their mouse away, allowing them to track their current position. The selectvalue layer will display the time step values at the end of the signal names, beside the start of the waveforms.

Next, to render the line for the time layer, we will create a plugin that will generate the line.

```
1 const pluginRenderTime = (desc, pstate, els) => {
2   const xmargin = 160;
3   const fontHeight = 20;
4   const fontWidth = fontHeight / 2;
5   const {height, xScale, xOffset, tgcd, timescale, xCursor} = pstate;
6
7   const body = [
8     // vertical line
9     ['line', {
10      class: 'wd-cursor-line',
11      x1: xmargin + 0.5,
12      x2: xmargin + 0.5,
13      y1: 0,
14      y2: height
15    }],
16   ];
17
18   els.time.innerHTML = stringify(genSVG(2 * xmargin, height).concat(body));
19 }
```

Next, we will create a mouseClickHandler that will handle the update of the time layer each time the user clicks on the waveform. We control the position of the line by simply setting the CSS left property based on some simple calculations. We know that the width from beginning of graph to the start of the waveform is 160px, which is fixed. We then obtain the X position of the user click, and store it for future use. We then subtract the two values to obtain the offset needed for the time layer.

```
1 const mouseClickHandler = (deso, els, pstate) => {
2   const { xOffset, xScale, tgcd } = pstate;
3
4   // width of waveql layer (from beginning to start of waveforms)
5   const xmargin = 160;
6   const handler = event => {
7     // get the selected time step x coordinate, and store it in pstate
8     const x = pstate.xTime = event.clientX;
9     // offset will be subtraction of the two distances
10    els.time.style.left = (x - xmargin) + 'px';
11  };
12  handler({clientX: pstate.xCursor});
13  els.view0.addEventListener('click', handler);
14  };
```

We then need to update the selectvalue layer with the values of the signal at that selected time step. This requires a deeper understanding on how VCDrom and its related tools work. VCDrom uses a [VCD parser](#) to parse the VCD file into a readable format. In particular, the parsed deso object contains a view and chango item that contains the data we need.

The view item contains a list of signals and its corresponding ref symbol. This ref symbol will be used to identify the signal in the chango item. An example of the view list is shown below:

```

1  [
2    {
3      "idx": 0
4    },
5    {
6      "idx": 1
7    },
8    {
9      "idx": 2,
10     "width": 8,
11     "name": "value",
12     "ref": "!"
13   },
14   {
15     "idx": 3,
16     "width": 1,
17     "name": "in_clk",
18     "ref": "\"\""
19   },
20   ...
21 ]

```

The chango item is an object containing each signal, storing the value of the signal waveform. The value of the waveform is stored as an array, and each item in the array indicates the value at a specific time step. For example, [17, 0] would indicate a value of 0 at time 17, which will stay at 0 until time 35 which it will go to 1, as indicated by [35, 1]. An example of the chango object is shown below:

```

1  {
2    "$": {
3      "wave": [
4        [0, 0, 1],
5        [17, 0],
6        [35, 1],
7        [45, 2],
8        ...
9      ],
10     "kind": "vec",
11     ...
12   },
13   "#": {
14     "wave": [
15       [0, 0],
16       [17, 1],
17       [28, 0],
18       [57, 1],
19       [68, 0]
20     ],
21     "kind": "bit",
22     ...
23   }
24   ...
25 }

```

With this information, we can loop through each signal in the view list, obtain the reference symbol and [calculate](#) the time step that the user selected. We then filter the wave to find the value of the signals where the time step is less than the selected time step.

```

1  const updateValueCol = (deso, els, xTime, pstate) => {
2    var values = [];
3    // loop through each signal

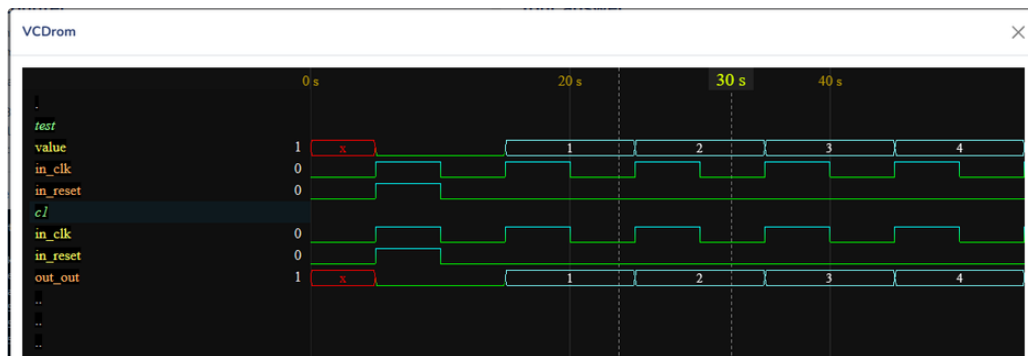
```

```

4  deso.view.forEach((item) => {
5      // get the reference symbol
6      const ref = item.ref;
7
8      if (ref) {
9          // calculate selected time step
10         const xx = ((pstate.xTime - pstate.xOffset) / pstate.xScale);
11         // filter the wave to obtain the value closest to the time step
12         const matchingWaveValues = deso.chango[ref].wave.filter((wave) => wave[0] <= xx).at(-1);
13         // check if the value is not X
14         if (matchingWaveValues && !matchingWaveValues[2]) {
15             values.push(['span', {class: 'wd-value'}, matchingWaveValues[1].toString(16)];
16         } else {
17             values.push(['span', {class: 'wd-value'}, 'X']);
18         }
19     } else {
20         // do not show any value for non-existant signals
21         values.push(['span', {class: 'wd-value-hidden'}, 'X']);
22     }
23 });
24 els.selectvalue.innerHTML = stringify(['div', {class: 'wd-valuecol'}].concat(values));
25 };

```

The image below shows the modified VCDrom with the new time step values feature:



Modified VCDrom with time step values

Control Buttons

VCDrom handles the various user interactives using keyboard / mouse shortcuts, shown in a menu. This will not fit AASP's use case for multiple reasons, with intuitiveness as the main concern. VCDrom's help menu is shown below:



VCDrom help menu

Instead, we will be implementing an array of interactive buttons that will replace these shortcuts. First, we will need to [render](#) the control buttons, which we will replace the menu layer with.

```

1  const renderControl = () => {
2    const zoomIn = ['button', {id: 'zoom-in', class: 'icon-button'}, '<i class="fa fa-search-plus icon-style"></i>'];
3    const zoomOut = ['button', {id: 'zoom-out', class: 'icon-button'}, '<i class="fa fa-search-minus icon-style"></i>'];
4    ...
5
6    const controls = ['div', {class: 'controls'}, zoomIn, zoomOut, ...];
7
8    return stringify(controls)
9  }
10
11 container.elo.menu.innerHTML = renderControl();

```

Next, we will need to map each button to its corresponding shortcut. VCDrom uses the [keyBindo](#) module to map a shortcut string to its corresponding action, so we will simply be executing these actions by adding an event listener to each of the buttons.

```

1  const controlMap = {
2    'zoom-in': 'zoomIn',
3    'zoom-out': 'zoomOut',
4    ...
5  }
6
7  const getControl = (pstate, els, render, cm) => {
8    Object.keys(controlMap).forEach((key, value) => {
9      document.getElementById(key).addEventListener('click', () => {
10        // execute the keyBindo action
11        if (executeControlHandler(controlMap[key], keyBindo, pstate, els, cm))
12          render();
13      });
14    })
15  }

```

```

16
17 const executeControlHandler = (key, keyBindo, pstate, els, cm) => {
18   if (key === 'zoomIn' || key === 'zoomOut') {
19     pstate.xCursor = pstate.xTime;
20     return keyBindo[key].fn(pstate, cm);
21   }
22   else
23     return (keyBindo[key] || keyBindo.nop).fn(pstate, cm);
24 };

```

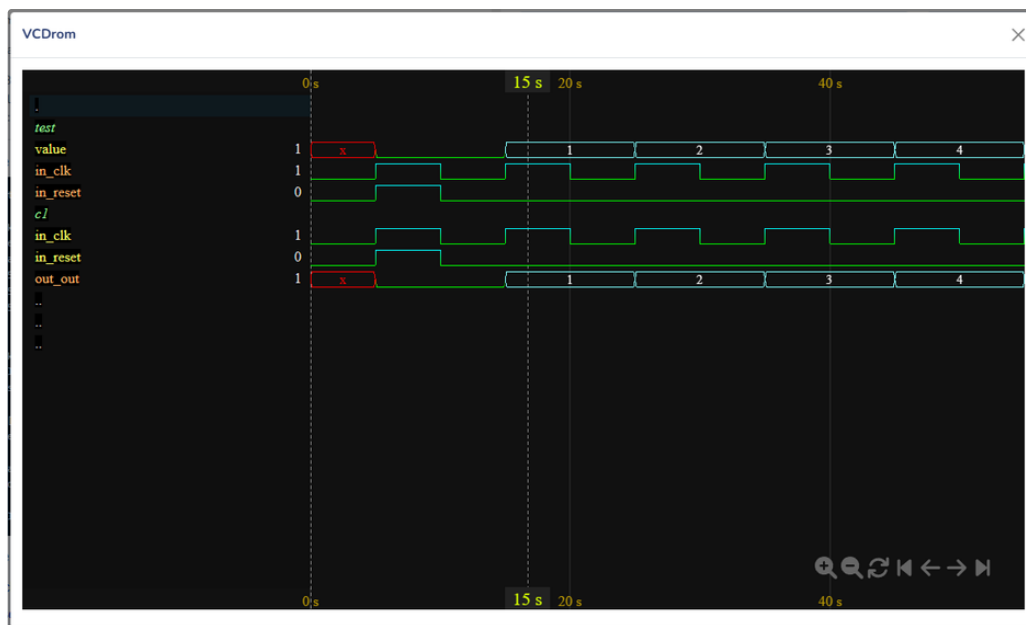
Lastly, we will need to update the selectvalue column each time there is a button interaction since it may change the selected time step.

```

1 const handler2 = () => {
2   updateValueCol(deso, els, pstate.xTime, pstate)
3 }
4
5 document.getElementById('zoom-in').addEventListener('click', handler2);
6 document.getElementById('zoom-out').addEventListener('click', handler2);
7 ...

```

The image below shows the modified VCDrom with the new interactive control buttons feature:



Modified VCDrom with control buttons

Build

To use VCDrom in AASP, we will need to build and copy over the files generated into the static and template folders in AASP. The [script](#) below will perform these steps:

```

1 #!/usr/bin/bash
2
3 mkdir -p ../static/vcdrom
4
5 cp node_modules/vcd-stream/out/vcd.wasm ../static/vcdrom
6 cp src/*.css ../static/vcdrom
7 cp src/*.woff2 ../static/vcdrom
8

```



```
9  mkdir -p ../templates/vcdrom
10
11  cp src/vcdrom.html ../templates/vcdrom/vcdrom.html
12
13  ./node_modules/.bin/browserify ./lib/vcdrom.js | ./node_modules/.bin/terser --compress -o ../static/vcdrom/vcdro
```

AASP Integration

Overview

This section will cover the integration of the various features in previous section into AASP, as well as cover some modifications done to implement HDL assessment support to AASP.

 This documentation assumes that you are proficient with the [Django Web Framework](#), the core framework AASP is built on.

This section will cover the following:

Judge0 Integration

The steps required to submit a Verilog HDL program to Judge0 for compilation is as follows:

1. Append or replace the \$dumpfile and \$dumpvars task to ensure VCD dump
2. Embed in_ and out_ to all port names
3. Create zip archive with all relevant metadata
4. Call Judge0 submissions API with the encoded zip archive as a parameter

 To understand why each of these steps are performed, refer to the implementation section for more details.

Inject \$dumpfile task

The [code](#) below looks for any existing \$dumpfile declarations, and if found, will replace the declarations to match what AASP expects. Otherwise, it will append the declaration to the last line of the testbench.

```
1 if testbench.find('$dumpfile') == -1:
2     # add wave dump to last line before endmodule
3     testbench = testbench.replace('endmodule', 'initial begin $dumpfile("vcd_dump.vcd"); $dumpvars(0); end endmodule')
4 else:
5     # Define the regular expression patterns
6     dumpfile_pattern = r'\$dumpfile\("[^"]+"\)'
7     dumpvars_pattern = r'\$dumpvars\(\d+\)'
8
9     # Replacement strings
10    new_dumpfile = '$dumpfile("vcd_dump.vcd")'
11    new_dumpvars = '$dumpvars(0)'
12
13    # replace wave dump
14    testbench = re.sub(dumpfile_pattern, new_dumpfile, testbench)
15    testbench = re.sub(dumpvars_pattern, new_dumpvars, testbench)
```

Embed in_ and out_

The embedding is done to both the [module](#) and [testbench](#) code. To find the input and output ports, we simply look for the lines starting with 'input' and 'output' in the module code, since this is the syntax for Verilog. We then append the in_ and out_ prefix to the input and output port names, and replace all existing declarations with the embedded version. To simplify the process for embedding the testbench code, we return the list of input and output ports, and [pass](#) to the testbench embedding method. With this, we can skip the searching process and directly embed the input and output ports for the testbench code.

Create zip archive

The [code](#) to create the submission file is shown below. We first write the module and testbench code into two separate files, 'main.v' and 'testbench.v' respectively. We will also need to specify a compile and run file to inform Judge0 how to compile and run our program using iVerilog. The compile command specifies the iverilog command and the two code files. The compilation output is then sent to [vvp](#), iVerilog's simulation engine. This will generate the VCD file, which we will pass to the vcd2wavedrom script. Doing so will print the WaveDrom text on its stdout, which Judge0 will recognize as the solution that will be used to compare with the expected solution.

```
1 with zipfile.ZipFile('submission.zip', 'w') as zip_file:
2     zip_file.writestr('main.v', module)
3     zip_file.writestr('testbench.v', testbench)
4     zip_file.writestr('compile', 'iverilog -o a.out main.v testbench.v')
5     zip_file.writestr('run', "vvp -n a.out | find -name '*.vcd' -exec python3 -m vcd2wavedrom.vcd2wavedrom --aasp -
```

We will also need to encode the zip archive in base64, following Judge0's [specifications](#).

```
1 # encode zip file
2 with open('submission.zip', 'rb') as f:
3     encoded = base64.b64encode(f.read()).decode('utf-8')
```

Call Judge0

We now have to append this encoded file to our parameters, and send to the Judge0 submission API for compilation.

```
1 # judge0 params
2 params = {
3     "additional_files": encoded,
4     "language_id": request.POST.get('lang-id'), # 90 for Verilog
5 }
6
7 # call judge0
8 try:
9     url = settings.JUDGE0_URL + "/submissions/?base64_encoded=false&wait=false"
10    res = requests.post(url, json=params)
11    data = res.json()
```

Static Waveform Integration

WaveDrom provides a CDN to embed timing diagrams into our AASP templates. The steps required to embed timing diagrams in AASP, following the [guide](#) outlined by WaveDrom, is as follows:

1. Add the [CDN](#) for WaveDrom
2. Set the [onload event](#)
3. Insert the WaveDrom [script](#) into AASP

Update Waveform

For AASP, we will need to [update](#) this waveform each time the user regenerate their output in the test case creation page. To do so, we will need to know the index of the waveform that we want to update, since there can be multiple test cases, each containing a waveform. WaveDrom automatically assigns an index to each WaveDrom script initialized, starting from 0. Thus, we can use the index assigned to each test case to identify the script, but we must take note to [update](#) the WaveDrom index as well when handling the deletion of a test case, which is shown below:

```

1 $(tcRow).find('[id^="WaveDrom_Display_"]').each(function() {
2   var id = $(this).attr('id').replace("WaveDrom_Display_", "");
3
4   // WaveDrom initializes two different elements that must be updated
5   $(tcRow).find(`[id="WaveDrom_Display_${id}"]`).attr('id', `WaveDrom_Display_${index}`);
6   $(tcRow).find(`[id="InputJSON_${id}"]`).attr('id', `InputJSON_${index}`);
7 });

```

We first obtain the WaveJSON from Judge0, and remove the embeddings.

```

1 let wavedrom = JSON.parse(res.data.stdout)
2 wavedrom.signal = wavedrom.signal.map(signal => {
3   const nameWithoutPrefix = signal.name.replace(/^in_|^out_/, "");
4   signal.name = nameWithoutPrefix;
5   return signal;
6 });

```

Next, we update the script value in step 3 with the new WaveJSON.

```

1 const e = document.getElementById(`InputJSON_${index}`);
2
3 // Update the input element's value with the wavedrom value
4 e.innerHTML = JSON.stringify(wavedrom);

```

Lastly, we must inform WaveDrom to re-render or reinitialize the waveform, using the [built-in functions](#) provided.

```

1 WaveDrom.RenderWaveForm(index, WaveDrom.eva(`InputJSON_${index}`), "WaveDrom_Display_")

```

Interactive Waveform Integration

AASP's VCDrom [API](#) requires a VCD file to be passed as one of its parameters, which will render the VCDrom html using the provided VCD file. For AASP, rather than displaying VCDrom on the same page as the question attempt page, we want to display it as a pop out modal.

This will be done using a iframe element:

```

1 <div class="modal-body">
2   <iframe id="modalIframe" width="100%" height="600" frameborder="0"></iframe>
3 </div>

```

When the user clicks on the 'Run' button, we will call the VCDrom API and pass the VCD file which we have stored in the button, and update the iframe with the rendered VCDrom.

```

1 const vcd = waveformBtn.data('vcd');
2
3 $.ajax({
4   type: 'POST',
5   url: `${url}vcdrom %}`,
6   data: {vcd: vcd, csrfmiddlewaretoken: `${csrf_token}`},
7 }).done((res, textStatus, jqXHR) => {
8   var iframeDocument = modalIframe.contentDocument || modalIframe.contentWindow.document;
9   iframeDocument.open();
10  iframeDocument.write(res);
11  iframeDocument.close();
12  vcdromModal.modal('show');
13 })

```

 The VCD file is [stored](#) in the button each time we compile the program.

Ace Editor

[Ace editor](#) is the code editor library used by AASP. However, some issues were faced when attempting to integrate HDL test case creation into AASP. In particular, HDL test case creation required multiple ace editors to be initialized, from the solution editors, to the test case editors.

When adding a test case, we will need to initialize the editor, which can be identified using the index assigned to the test case.

When removing a test case and updating the IDs of the test case rows, it created multiple bugs that rendered the editor unusable. For example, the editors with updated IDs would update the existing code to a bunch of Xs. The solution I found involved destroying the editor and reinitializing it.

```
1  const destroyEditor = (index) => {  
2    const editor_id = `id_tc-${index}-stdin-editor`;  
3    ace.edit(editor_id).destroy();  
4  }
```
