Pinfile Builder Documentation

Adam Ibrahim

Use

This is a program for quickly creating pin assignment files for use with the Altera DE-2 board and Quartus II's FPGA programming software.

If you've used it, you know that specifying the actual pin name for each signal in a file seems like it should be a much easier task.

Features

- You can use the descriptive pin names that are written on the board, which are also listed in the pin assignment file.
- You can specify assignments for *ranges* of signals and *ranges* of pins.
- By altering the `pin_table.json` file, you can create more convenient names for pins.
  - One such mapping is already provided for you: You can assign signals to more than one seven segment display at a time by using the descriptive name `HEX`, instead of `HEX0`, `HEX1` and so on. For examaple:
  - `HEX[0..6]` is the same as `HEX0[0..6]`
  - `HEX[7..13]` is the same as `HEX1[0..6]`,
  - `HEX[14..20]` is the same as `HEX2[0..6]`, et cetera.
- Writes the header for your pin file for you (TO, LOCATION).

Example

Normal Pin Assignment File:

```
TO, LOCATION
output[0], PIN_AE23
output[1], PIN_AF23
output[2], PIN_AB21
output[3], PIN_AC22
output[4], PIN_AD22
output[5], PIN_AD23
op0[0], PIN_N25
op0[1], PIN_N26
op0[2], PIN_P25
op0[3], PIN_AE14
op0[4], PIN_AF14
op0[5], PIN_AD13
op1[0], PIN_AC13
op1[1], PIN_C13
op1[2], PIN_B13
op1[3], PIN_A13
op1[4], PIN_N1
op1[5], PIN_P1
opcode[0], PIN_U4
```

```
opcode[1], PIN_V1
opcode[2], PIN_V2
execute, PIN_G26
```

That's quite long! You can shorten all of that to:

Shorthand pin assignment file:

```
output[0..5], LEDR[..]
op0[0..5], SW[..]
op1[0..5], SW[6..]
opcode[0..2], SW[15..]
execute, KEY[0]
```

The latter is easier to maintain, since you can change a lot of assignments quickly. It's also more descriptive. You can come back to an older project and know what to expect when you put your program on a board.

How to use

There are two major programs here: - **pin_parser.js**, which transforms a file that's in the descriptive shorthand demonstrated above and unwraps it such that there's only one index in between each set of square braces. - Use: - `node pin_parser.js` : Will take its input from stadard input and write to standard output. - `node pin_parser.js path/to/inputFile` : Will take its input from `inputFile` and write output to standard output. - `node pin_parser.js path/to/inputFile path/to/outputFile` : Will take its input from `inputFile` and write output to `outputFile`. - **replacer.js**, which takes an unwrapped shorthand file and translates it into a file of actual pin names. - Use: `node replacer.js` : Will take its input from standard input and write to standard output.

Specification of Shorthand

Each shorthand line has the following form:

```
name[ index_specifier ], name[ index_specifier ]
```

Where the name to the left of the main comma is a signal name, and the name on the right is a descriptive name of a pin, or set of pins such as `SW`, or `HEX0`, or `CLOCK_27`.

Example:

```
output[0..5], LEDR[..]

{ output } [ {       0..5      } ], { LEDR } [ {        ..       } ]
{  name  }   { index_specifier }    { name }   { index_specifier }
```

The result of the above shorthand line is to assign LEDR[0] to output[0], LEDR[1] to output[1] and so on.

Index Specifiers

An **index specifier** is a shorthand for a list of indices. For example, `0..5` is a short way of saying `0,1,2,3,4,5`.

There are two major types of index specifiers: **explicit specifiers** and **implicit specifiers**. Plainly speaking, an explicit specifier is a specifier that gives enough information to go from a shorthand for a list of indices straight to the list of indices. `0..5` is explicit because I know that it includes every integer in between 0 and 5 inclusive.

An *implicit specifier* is a specifier that does not give enough information to go from a shorthand list to a full list. They require a little bit of context to understand what they mean. `..` is an example of an implicit specifier. The extra information needed for an implicit specifier is provided by an explicit specifier on the same line.

So the meaning of the example line is: Use indices `0,1,2,3,4,5` of the signal output, and use the exact same index list for RED LED's.

Syntax Rules

- Non-blank lines must contain at least one comma in order to separate the signal name from the pin name. This comma should be after the end of the signal name (including an index for that signal, if it has one) and before the start of the pin name.
- Blank lines are allowed.
- Index specifiers must be placed in between square brackets.
- An index specifier must match a known format of index specifier, which are listed below in the two index specifiers section.
- Both index specifiers must have the same length.
  - There is no point in trying to make them have differing lengths since:
    * For each signal name in the file, there has to be a pin assigned to it.
    * If we tried to re-use pin/signal pairs, the last read pair takes effect over all previous pairs. New assignments overwrite old assignments.
- For every line, at least one index specifier must be explicit.

Explicit Index specifiers

These are index specifiers that can be immediately resolved. No extra information is necessary to figure out the indices that are specified.

- Simple Explicit specifier :
  - Format : `A,B,C,..`
    * A,B,C… : Each of these is an integer.
  - Comma separated list of at least one index.
  - Example : `2,3,5,7,11,13`.
- Simple Range :
  - Format : `A..B`
    * A: Start
    * B: End
  - Specifies an inclusive range of consecutive integers. Begins with 'A' and ends with 'B'.
  - Example : `3..5` -> `3,4,5`
- Step To End :
  - Format : `A:B..C`
    * A: Start
    * B: Step Amount
    * C: End
  - Specifiers a sequence of indices which are separated by a common difference, or step amount (an arithmetic progression). Sequence starts at 'start' and proceeds to the largest number of the form 'start + k*step' that is less than or equal to 'end'. In other words, A is a start, C is an end, and B is a step amount.
  - Example : `2:3..10` -> `2,5,8`
  - Example : `2:3..11` -> `2,5,8,11`
- Step Range :
  - Format : `A:B:C`
    * A: Start
    * B: Step Amount
    * C: Length Control
  - Similar to 'Step to End' specifier. Specifies a sequence of indices that starts from the 'start', uses 'step' as a common difference. However, rather than proceeding while it does not exceed an end, it creates a sequence with length 'length + 1' such that:

    * The first number is ‘start’.
    * The last number is ’start + length*step’.
  - Example: `2:3:5 -> 2,5,8,11,14,17`

Implicit Index Specifiers

These index specifiers omit some information about which indices to specify. Since each line must include a “TO” entry and a “LOCATION” entry, an implicit specifier can use the indices from an explicit index specifier on the same line to build an index list.

This means that there must be at least one explciit index specifier on every line.

> NOTE
>
> Implicit index specifiers are not limited to ’TO’ entries. They can also be used in ’LOCATION’ entries.

- Range Reuse:
  - Format : `..`
  - Uses the same index specifier as an explcit index specifier.
  - Example:
    ```
    Implicit Form : Array[..]   , SW[0..5]

    Explicit Form : Array[0..5] , SW[0..5]
    ```
- Offset Range Reuse:
  - Format : `A..`
    * A: Offset
  - Reuses the indices specifies by the explicit specifier, but adds a common offset to all of them.
  - Example:
    ```
    Implicit Form : array[2..]  , SW[1..6]

    Explicit Form : array[3..8] , SW[1..6]
    ```
- Length Forward :
  - Format : `A..#`
    * A : Start
    * # : A + length - 1
  - You can think of the ‘#’ as being a placeholder forthe length of the explicit specifier index list. So this acts like a simple range, which starts atstart’ and ends at ‘start + length - 1’. Thelength - 1’ is to ensure that both the explicit specifier and the implicit specifier are the same length.
  - Example:
    ```
    Implicit Form : array[5..#], SW[1..6]

    Explicit Form : array[5..10], SW[1..6]
    ```
  - Example:
    ```
    Implicit Form : array[5..#], SW[0:2..10]

    Explicit Form : array[5..10], SW[0:2..10]
    ```
  - Exmaple:
    ```
    Implicit Form : array[5..#], SW[2,3,5,7,11]

    Explicit Form : array[5..9], SW[2,3,5,7,11]
    ```
- Length Backward
  - Format : `#..A`
    *

: A - (length - 1)

    * A : End
  - You can think of the '#' as being a placeholder for the length of the explicit specifier index list. So this acts like a simple range, but reversed. It starts at 'end' and proceeds backward toend - length + 1'. The 'length+1' is there to ensure that the explicit specifier and the implicit specifier are the same length.
  - Example:
    ```
    Implicit Form : array[#..8], SW[1..6]

    Explicit Form : array[3..8], SW[1..6]
    ```
  - Example:
    ```
    Implicit Form : array[#..8], SW[0:2..10]

    Explicit Form : array[3..8], SW[0:2..10]
    ```
  - Exmaple:
    ```
    Implicit Form : array[#..8], SW[2,3,5,7,11]

    Explicit Form : array[4..8], SW[2,3,5,7,11]
    ```
- Step Range Length Reuse :
  - Format : `A:B:#`
    * A : Start
    * B : Step
    *

: length - 1

  - You can think of the '#' as a placeholder for the length of an explicit specifier. So the index list produced by this specifieris exactly like the step range explicit specifier, with the 'length' parameter replaced by the length of the explicit specifier, minus 1.
  - Example:
    ```
    Implicit Form : array[0:2:#]     , SW[0..5]

    Explicit Form : array[0:2:4]     , SW[0..5]

    Explicit Form : array[0,2,4,6,8] , SW[0..5]
    ```
  - Example:
    ```
    Implicit Form : array[1:2:#]   , SW[0:3..10]

    Explicit Form : array[1:2:3]   , SW[0:3..10]

    Explicit Form : array[1,3,5,7] , SW[0:3..10]
    ```
  - Exmaple:
    ```
    Implicit Form : array[4:3:#]           , SW[2,3,5,7,11,13]

    Explicit Form : array[4:3:5]           , SW[2,3,5,7,11,13]

    Explicit Form : array[4,7,10,13,16,19] , SW[2,3,5,7,11,13]
    ```

How to extend this program

Where Specifers are Defined

All specifier information is kept in `formats.js`. The information is split into two tables per specifier: a formats table and a resolution table. The formats table contains regular expression objects which both identify and parse specifiers, and the resolution table accepts a parsed expression (in the form of a Regular Expression object), reads the information from it, and produces an array which contains the indicies specified.

Adding New Specifiers

If you would like to add new specifiers of your own to those presented here, then you must know the following:

- Every specifier has a format in the formats table and a resolving function in the resolution table.
- For each specifier, the entries for that specifier in each table must have the same key (name). So, for example, with the 'Explicit Simple Specifier', there are two entries in the tables for explicit specifiers:
  - explicitSpecifierFormats.simpleExplicit
  - explicitResolution.simpleExplicit
- The formats for each specifier are regular expressions.
- Explicit and Implicit specifier information is kept separate. There are two tables for each kind of specifier:
  - Formats Table: A table of regular expressions which match and parse the specifier.
  - Resolution table: A table of functions which accept formats. In JavaScript, regular expression objects have a state based on their last match. They keep track of captured groups, so they're basically parsed forms of the specifier.

Program Stuff

Specifier Objects:

The have the following properties:

- content : The completely resolved array of indices.
- text : The actual specifier.
- Type : Implicit/Explicit (either 'e' or 'i')
- Name : specifier name, such as "simpleExplicit"
- match : the result of a format match.
- Length : The length of the completely resolved array of indices. This is useful for resolving implicit specifiers.

Line form:

```
Vim Specific Regex: \v(\w+)\[([^]]+)]\s*,\s*(\w+)\[([^]]+)]

(\w+)\[([^]]+)]\s*,\s*(\w+)\[([^]]+)]

( \w+     ) "[" ( [^]]+              ) "]," ( \w+           ) "[" ( [^]]+                    ) ]
( TO_name )     ( TO_index_specifier )      ( LOCATION_name )     ( LOCATION_index_specifier )
```

Recommendations

- If on one line, one index specifier is related to another, try to make use of implicit specifiers. They reduce the amount of work you need to do, and they keep your pinfile easier to update, easier to read, and easier to understand. They make your intent for assignment more clear.
  - Example:
    * Don't: `signal[0..10], pin[0..10]`

* Do: `signal[0..10], pin[..]` or `signal[..], pin[0..10]`
* The use of the range reuse implicit specifier tells the reader that you want to assign the first 11 signals to the first 11 pins of some kind, rather than making the reader infer this by checking to see that both index specifiers are the same.

Coming Soon

Named Constants

Place named constants in a pinfile preamble in the following form:

`name=value`

The pinfile parser will scan for a preamble, create a list of these named constants, then replace each one with with the value that is associated with that name. After that, parsing will begin as normal.

Generating a .pinfile from VHDL source

Create a separate program to read in a VHDL file and generate an incomplete pinfile for that VHDL source. This partial pinfile will have: - Named constants for every port signal that is a vector: - name'left : The leftmost index of the vector. - name'right : The rightmost index of the vector. - name'length : The length of the vector, which is equal to - If vector declared using `downto`: name'left - name'right + 1. - If vector declared using `to`: name'right - name'left + 1. - A list of all port signals. Makes it very easy to type out the important parts of the pinfile. - Signals that are vectors will be listed as such: `vector_name[` - Signals that are nt vectors will be listed as such: `name,` # TODO

- Fix documentation to reflect that lengthBackward implicit specifier actually counts backward, so that A..# produces A, A+1, .., whereas #..A produced A, A-1, …
- Add a 'header' line format and skip it.
- Test header ignoring from .pinfile.
- Make lengthBackward extend backward, but count forward.
- Make reverses of common specifiers:
  - Reverse range reuse
  - Negative step values
  - Count downward for step ranges.
  - reverses for length reuse.
- Make simple variables through text substitution, like define macros in C/C++.
- Scan the ports area of an entity declaration in a VHDL file, then take one of two actions:
  - If there is no existing pinfile, then spit out a partial pinfile with defined variables for the starts, ends and lengths of various input vectors. Allow user to type out the pin assignments using those variables.
  - If there is an existing pinfile, then just update the variable preamble section with the current values for those variables. This way, the user can use pinfiles to set up general rules for pin assignments and not have to worry about them too much.