Modelling a Procedural Calculator Using Definitive Methods

0705846

Abstract

Calculators are very procedural tools, but at a component level, when dealing with boolean bits of data, they lend themselves very well to definitive modelling. Here a model calculator – constructed in Cadence – is introduced, which implements both the low-level bit operators and the high-level states associated with using a calculator, but using a definitive static-web architecture instead of a classic CPU-oriented procedural design. Such an architecture has the advantage of having no "clock-speed", because calculations are performed in a massively parallel style, with no need for a central control or iteration. This project assesses Cadence's applicability to modelling such systems.

1 Introduction

This project introduces a fully functional 8-bit signed integer[1] calculator created in Cadence, which rather than using a traditional CPU-oriented design uses a static web of connected logic gates. The model spans from the bit-level operations of addition, subtraction, multiplication and division to the high-level states associated with using a calculator, including entering numbers by digit, unary operators, binary operators and clear functions. The aim is to show how a definitive static-web approach can successfully be modelled using Cadence. A user interface has also been created for end users to make the model more visually appealing, and this can be seen in figure 1 (The included readme.txt file contains instructions for running this interface).

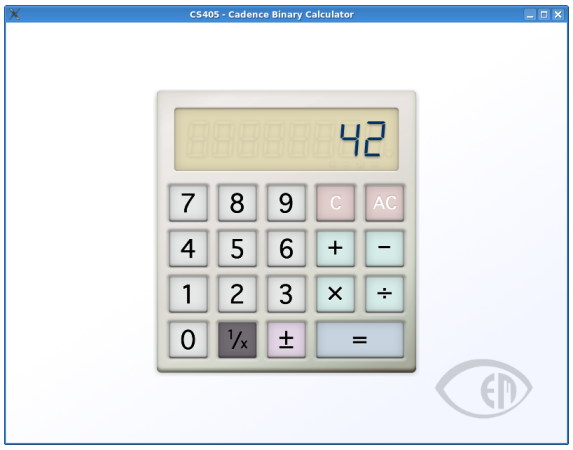

Figure 1: The user interface. The user has turned the calculator on and entered 42, with the sequence AC 4 2.

[1]Though described as 8-bit, technically the calculator is 9-bit; it contains 8 bits of precision and one bit for sign.

2 Implementation

Each of the connections, physically analogous to electrical pathways, are implemented using "is" definitions, resulting in an internal structure for the calculator which is static[2]. Such a static web is ideal for real-world manufacture, because it can be created as a simple electronic circuit composed of standard parts. It is also especially suited to Cadence's definitive structure.

2.1 High Level States

A calculator can exist in several states. In this design, the colour of each button relates to the associated state change: white will put the calculator into a number-input state, green into an operation state, blue into an answer state, red will reset certain stores, and purple will toggle flags for the current input. (Red and purple will also put the calculator into a number input state if necessary).

Internally the calculator contains two stores: input and stack. These can both be displayed on the LCD, and which one is shown depends on the current state[3]. An example sequence of interactions and their associated state changes is shown in Table 1.

Note that the ± button has no effect on state (though it will set the display to show the Input value if necessary), but will change a toggled flag determining how the input is displayed on screen (and used in

[2]The "willbe" (:=) definition has been required in one area; conditional bit stores, which must retain their value until told to change. These are in turn used to store user input. Willbe has also been used in some areas of the user interface, but this is outside the scope of the primary model.

[3]The input store is displayed when in a number input state, and stack otherwise.

Table 1: Intermediate internal states for a sequence of buttons.

Button	State	Op.	Input	Stack	Displayed	Notes
	Off		null	null	Input (null)	
AC	Input		0	0	Input (0)	
4	Input		4	0	Input (4)	
2	Input		42	0	Input (42)	When pressing any number, the input value is multiplied by 10 and the number is added.
×	Display	×	0	42	Stack (42)	Input has been copied into the stack. The operation state can be seen graphically as a small × symbol beneath the number in the LCD display.
2	Input	×	2	42	Input (2)	
−	Display	−	0	84	Stack (84)	This time the old state was multiply, so input × stack has been copied into the stack.
0	Input	−	0	84	Input (0)	
1	Input	−	1	84	Input (1)	
0	Input	−	10	84	Input (10)	
=	Display		0	74	Stack (74)	
÷	Display	÷	0	74	Stack (74)	Note that only change is the operation state. This is because the previous state was Display.
2	Input	÷	2	74	Input (2)	
=	Display		0	37	Stack (37)	
3	Input		3	0	Input (3)	

operations). The $[1/x]$ button has the same functionality, but is not used in this implementation due to a lack of fractional values, and for this reason is disabled in the interface.

It is also worth mentioning that the initial states for Input and Stack are null as a result of Cadence's way of working with "willbe" definitions; it is impossible to simultaneously set a willbe definition and set an initial value – there must be some delay between the commands. However, this is used to an advantage because it causes the screen to remain blank until these states are set to 0 with the AC button, hence simulating turning the calculator ON. These null values can perhaps be interpreted as the circuit having no power, and hence no fixed values of either true or false.

The high-level state can, to some extent, be seen graphically on the LCD; the current operation is displayed below the digits, and the overflow state is denoted by a dot to the right of the number.

2.2 Binary Operations

The low-level binary operations are modular collections of logic gates, which are used and re-used in many areas throughout the calculator. For example, several subtraction operators are used within the division operator, and several division operators are used when displaying a value on the LCD. Cadence's %deep cloning has proven invaluable here, as it allows easy re-use of structures in different situations.

Figure 2 shows an outline of the subtraction operator as an example of this structure. The AND, OR and NOT boolean gates are provided by default in Cadence, and the XOR gate was added as an extension (figure 3). Even here Cadence's %deep cloning is useful, as each bit-level subtraction (shown on the right of figure 2) is created from a single cloned object[4], significantly reducing the number of repeated operations which need to be directly implemented.

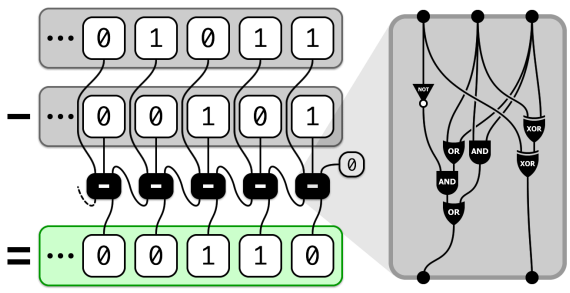

Figure 2: The subtraction operator shown as a collection of logic gates. The zoomed-in view on the right is contained within each of the black [-] boxes on the left.

[4]Each object references a bit from each input, and the previous result bit. The calculations for the output value and carry flag are inherited and operate on these inputs.

These low-level states are hidden from general users, but through Cadence's design are made available in the IDE (figure 4). This allows any user with some deeper knowledge of the model to directly modify any part of the calculator, for example by causing a bit to be stuck on, or modifying the behaviour of certain buttons to simulate a hardware glitch. The potential ways in which these bit operations might fail can be better understood in this way. Similarly, new functionality can be added (such as adding a collection of operators to create a square-root function), and "hidden" functionality can be revealed, for example the base of the number displayed on the LCD can be changed from 10 with the command ".rendererStuff lcdbase = (@staticBytes byte_16);"

```
1 true xor = (new
2     true = false
3     false = true
4 );
5 false xor = (new
6     true = true
7     false = false
8 );
```

Figure 3: Extending boolean logic to implement XOR in Cadence, in "cadenceExtensions.dasm".

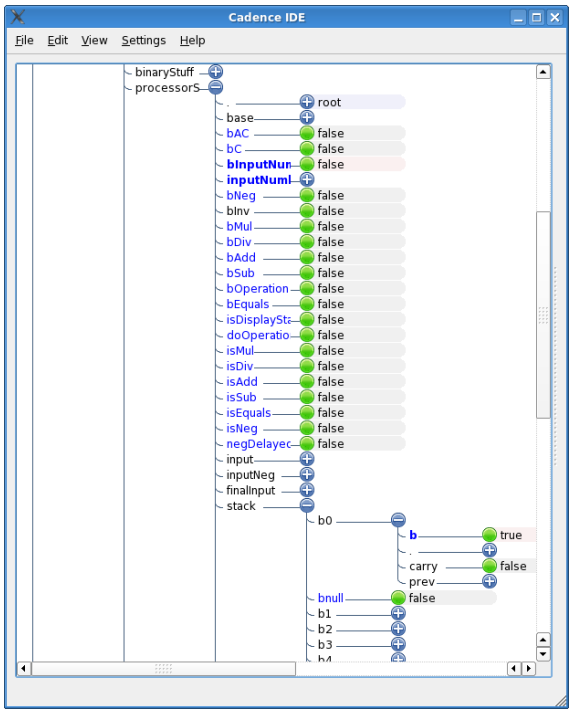

Figure 4: Cadence's variable viewer, showing many of the high-level states, and expanding the stack variable to show the state of its 2^0 bit (b0).

2.3 User Interface

The user interface has been created using the Warwick Game Design (WGD) library, through an existing wrapper for Cadence. It consists of many images, including a background, buttons, and each LCD segment. These images are swapped depending on the current state, for example when clicking on a button its image is replaced to show a pressed button.

Because of the very limited range of numbers supported, the overflow flag is given greater significance in the display than in a standard calculator. When a number overflows for any reason (which may be because of a divide-by-zero operation, or a simple addition / multiplication beyond the range of the calculator), a dot is displayed on the right of the display, and the displayed digits are replaced with the message "Err" for Error (figure 5).

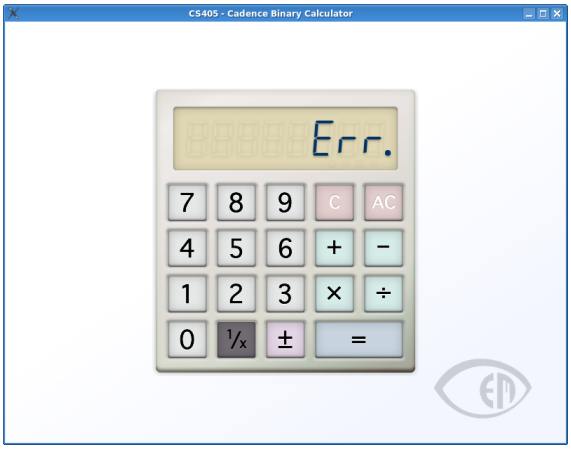

Figure 5: The result of a divide-by-zero operation; an error message.

The communication between the user interface and the model was designed to use simple and obvious handles, without any need for deep associations. This gives the advantage of portability with different displays, or even to some extent different processing logic. Cadence's "is" definition is very well suited to this, providing a natural and intuitive hook/handle syntax.

3 Analysis

The final model works well, but suffers from delays due to its size. The use of 16-bit calculations was explored[5], but the increased complexity was found to make the model very slow and highly unstable.

[5]Files extending the logic to 16-bit are included. See the readme.txt file for instructions on how to use them.

This experiment with 16-bit calculations also revealed an important limitation in Cadence; functions and iteration are not implemented. While neither of these entirely fit with the design of the language, their absence causes the need for large amounts of repeated code. For example, when extending the binary logic, each operation required (at least) an additional 8 lines of code to extend the operation to include the new bits of precision. With iteration or some similar principle, this could be reduced to changing a variable.

The LCD digits, being composed of 7 images, each have the ability to take one of 128 forms. Only a selection of these are realised, taking the appearance of the numbers 0 through 9 and letters A through F (for base 16 displays), and the lowercase letter r, seen only when displaying an error message (figure 5). This flexibility shows that the resulting variables do not need to be translated into Cadence's number system to be displayed; the same boolean logic used elsewhere in the design can be applied to the output number to determine which segments of the LCD are required. This is a very elegant result, because it frees the model from working with Cadence's more abstract pseudo-infinite number groups[6].

3.1 Limitations Encountered

During this project, a few limitations were encountered which required workarounds. The first was Cadence's handling of the willbe (:=) operation. Cadence does not currently allow the same variable to be given a willbe definition *and* an equals (=) definition in the same file; it requires some delay between the commands. While this was used to an advantage in this project, the limitation is important and should be addressed (note that any fix for the issue will *not* break this model, because the model makes no attempt to use such a feature).

Within the WGD wrapper, only images directly contained within the ".widgets root children" environment are displayed on the screen, and this has resulted in the need for many lines of code which simply add references to images defined elsewhere. The ability to display images which are indirectly contained in this environment would be more useful[7]. Alternatively a form of iteration as discussed above would help.

Finally the WGD library's use of lazy texture loading, and apparent lack of caching or re-use of loaded textures, was found to cause significant delays when updating the user interface. A workaround has been used in this model, where each image always exists and is visible, but displayed at a y coordinate of -100 when not desired. Obviously this is not ideal, and a texture caching mechanism would be better.

Also, while not a limitation, constructing the user interface is currently a laborious task of finding the correct coordinates to position individual elements. A useful extension or plugin to Cadence would be a user interface prototyping tool, able to generate code for constructing a particular interface which has been designed graphically, through more user-friendly point-and-click methods. Such a tool is suggested as another potential future development.

4 Conclusion

This project has revealed many aspects of Cadence's structure which lend themselves well to the modelling of static webs of operators, which has particular relevance to massively parallel circuit simulations. While several limitations were found, these are predominantly minor issues which can be easily fixed or worked around. The exception to this is the more significant issue of iteration, which does not exist within Cadence. This limitation requires many lines of code to be used where one should suffice, and is the single biggest factor limiting Cadence's applicability to the modelling of complex structures such as calculator circuits.

The final calculator model performs well, but its high complexity causes it to respond slowly to inputs. In theory the calculator could support 64-bit floating point data with some work (though the associated increased complexity would slow the interface down much further, as was seen when investigating 16-bit integers), but due to the limited time spent on its construction it is currently restricted to 8- or 16-bit signed integer operations.

Overall it is clear that Cadence is well suited to the modelling of complex *mostly static* circuits such as those introduced in the calculator model, but requires some work to improve the efficiency of certain tasks.

Bibliography

Nick Pope. PhD thesis, University of Warwick, 2011. (draft).

Dawn Rorvik. Binary Arithmetic – Technical Notes (Binary Division section). Online, May 2003. http://academic.evergreen.edu/projects/biophysics/technotes/misc/bin_math.htm Accessed Jan. 2011.

[6] Cadence's number system is still required in the construction of the user interface, due to the WGD library's implementation.

[7] Currently 14 lines of code are required to display each LCD digit. With indirect inclusion this could be reduced to one.