CAL on CHS2x4 Boards Made Simple

Gordon Brebner
Department of Computer Science
University of Edinburgh
James Clerk Maxwell Building
Mayfield Road
Edinburgh EH9 3JZ
Britain
(E-mail: gordon@dcs.ed.ac.uk)

1. Introduction

This document is a short guide to the design and exercise of Configurable Array Logic (CAL) circuits using the Algotronix CHS2x4 board for IBM PCs (or clones). It describes some additional software which is meant to enable a designer with a software background to make rapid progress without becoming bogged down in too many technical details.

It is assumed that the reader is familiar with the nature of CAL technology: a basic architecture which is a programmable cellular array with nearest neighbour connections. Cells and connections are very simple, supporting only single-bit operations and transfers. Cells have a function unit, together with communication points on all four sides; also, G_1 and G_2 are two global signals which are distributed to all cells in the array. The function unit can be programmed either to perform one of the 16 possible two-input Boolean operations, or to act as a clocked one-bit latch; if it is not programmed, the cell only through-routes connections. Within-cell routing can be programmed in order to interconnect the function unit inputs and output, and the perimeter communication points.

The CHS2x4 board is meant to contain a 2x4 array of CAL1024 chips, which gives a 64x128 array of CAL cells, but most current boards contain only a 2x3 array of chips, giving a 64x96 array of cells; this latter arrangement will be assumed in this document. The board also contains an additional CAL1024 chip which controls the operation of the main CAL array. The standard configuration of this control circuitry allows the array to be exercised in a simple fashion: a program running on the PC can cause eight single-bit values to be input to, and eight single-bit values to be output from, cells on the perimeter of the array, during a short period (around 100 ns) while the global signal G_1 is held high. This mechanism hints that G_1 should be reserved for timing in CAL circuit designs.

Section 2 of this document describes an appropriate CAL design style which uses global signal G_1 consistently with the CHS2x4 board. Section 3 describes a simple programming interface to allow CAL circuits to be created. Section 4 describes some tools for handling CAL circuits before they are exercised. Section 5 describes a simple programming interface to allow CAL circuits to be exercised on the CHS2x4 board. Section 6 outlines the availability of the extra CAL software. An appendix contains a design example.

2. Recommended circuit design style

Given a central clocking rôle for global signal G_1, it is useful to employ latches clocked by G_1 to play the following parts in the circuit:

- input latches clocked by G_1;
- output latches clocked by $\overline{G_1}$; and
- storage flip-flops of master-slave latch pairs clocked by $\overline{G_1}$ and G_1 respectively.

Then, the operation of the circuit can be made synchronous and sequential.

The circuit can be considered as having a sequence of operational steps, each with two phases. During the first phase, when G_1 is high, input and output is performed: input values are stored in input latches and output values are available from output latches; also, internal master to slave transfers take place in storage flip-flops. During the second phase, when G_1 is low, computation is performed: combinational logic transforms data from input latches and from storage flip-flop slave latches to produce results in output latches and in storage flip-flop master latches.

The global signal G_2 is available for general-purpose use. Given the above regime, G_2 would not be expected to have any rôle during the input and output phase, but it can be of use during the computational phase. For example, it may be convenient to use G_2 to provide global initialisation for the storage flip-flops.

The layout of the circuit is constrained by the position of the cells at which input and output values can be supplied on the CHS2x4 board. The eight single-bit input values, in the order least to most significant bit of a single byte supplied by the PC, are available *both* to the left-hand (west) side of cells

(0,16), (0,18), (0,20), (0,22), (0,24), (0,26), (0,28), (0,30)

and to the right-hand (east) side of cells

(95,32), (95,36), (95,40), (95,44), (95,48), (95,52), (95,56), (95,60).

The eight single-bit output values, in the order least to most significant bit of a single byte returned to the PC, are available *either* from the left-hand (west) side of cells

(0,0), (0,2), (0,4), (0,6), (0,8), (0,10), (0,12), (0,14)

or from the right-hand (east) side of cells

(95,0), (95,4), (95,8), (95,12), (95,16), (95,20), (95,24), (95,28).

(If the board contains a 2x4 array of cells, "95" should be replaced by "127" in the above.) Input and output latches must be placed reasonably close to these positions, since the duration of the input and output phase (i.e., the time G_1 is high) is relatively short.

3. The CLAP library

CAL circuits can be created using the graphical CLARE tool or the C program-based LAURA library, both from Algotronix. The CLAP library is an alternative C program-based library which is simpler than LAURA. The aim is for a program to produce a CAL 'block', that is, a circuit consisting of a rectangular array of CAL cells with input and/or output points around the perimeter. To start creating a block, the program calls:

```
begin_block (name)
```

where the `name` parameter is a string to be used as the name of the block. To finish creating a block, the program calls:

```
end_block ()
```

and a description of the block is stored in Algotronix CFG format in a file named by the block name suffixed by "`.cfg`". Between these two calls, other functions can be called to describe input/output points, to describe individual cells, or to describe sub-blocks. The finished block is defined to be the smallest rectangular array which contains all of the described components. Obviously, it should fit within the confines of the CHS2x4 array.

To describe an input or output 'port' of the block (that is, a location at which input or output values are supplied to or from the circuit), the program calls:

```
block_port (name, way, x, y, loc)
```

where the `name` parameter is a string to be used as the name of the port. The `way` parameter is of type `port_type`, and is equal to `in_port` for an input port or `out_port` for an output port. The `x` and `y` parameters are integers, and give the coordinates of the cell to which the port is attached; the `loc` parameter is of type `cell_location_type`, and is equal to `top`, `bottom`, `left` or `right` indicating which side of the cell the port is attached to. Normally, `way`, `x`, `y` and `loc` will collectively correspond to one of the 32 CHS2x4 input/output points which are listed in Section 2.

To describe an individual cell of the block, there are three stages. To start describing a cell, the program calls:

```
begin_cell (fn, x, y)
```

where the `fn` parameter is of type `cell_function_type`, and is equal to one of 21 possible values indicating the function performed by the cell function unit:

- no function: `wire`;
- constant value: `zero` or `one`;
- unary operator: `a_ident`, `not_a`, `b_ident` or `not_b`;
- binary operator: `a_and_b`, `a_and_not_b`, `not_a_and_b`, `a_nor_b`, `a_or_b`, `a_or_not_b`, `not_a_or_b`, `a_nand_b`, `a_xor_b` or `a_xnor_b`;

- C-D latch: `c_d_latch`, `not_c_d_latch`, `c_not_d_latch` or `not_c_not_d_latch`

where `a` and `c` are 'left-hand' function inputs, `b` and `d` are 'right-hand' function inputs, and the operation names are mostly self-explanatory (`ident` is the identity operation, and `not` denotes negation of a function input). The `x` and `y` parameters are integers, and give the coordinates of the cell. To finish describing the cell, the program calls:

```
end_cell ().
```

Between these two calls, to describe intra-cell wires, the program can call:

```
from_to (f, t)
```

where the `f` parameter is of type `cell_location_type`, and is equal to `top`, `bottom`, `left`, `right`, `g1`, `g2` or `f_out`, indicating the four sides of the cell, the two global signals, or the function unit output, respectively as the source of the wire. The `t` parameter is of type `cell_location_type`, and is equal to `top`, `bottom`, `left`, `right`, `a_in/c_in` or `b_in/d_in`, indicating the four sides of the cell, or the two function inputs, respectively as the destination of the wire. Although CLAP does not check, CAL disallows more than one wire to the same destination, global signals wired to the right-hand function inputs `b_in` and `d_in`, and the function unit output wired to function unit inputs.

It is common to need wiring cells which have a function unit performing no function and a single intra-cell wire. As shorthand in such cases, the program can just call:

```
wire_cell (f, t, x, y)
```

where the parameters `f`, `t`, `x` and `y` are as for `begin_cell` and `from_to` above.

CLAP is intended to produce a 'flat' block (that is, with no hierarchy), because structuring and repetition of components can be reflected in the C program rather than in the final circuit. However, to allow the inclusion of library blocks, or of blocks created using CLARE, it is possible to refer to instances of other blocks as 'sub-blocks' of the main block. To describe a sub-block, the program calls:

```
sub_block (name, trans, x, y)
```

where the `name` parameter is a string giving the name of the block to be included. To check its existence, this block is looked up in a file named by its name suffixed by "`.cfg`", in the current directory or the directory determined by the environment variable `CFG_FILE_PATH`; any sub-blocks of this block are also read from appropriately-named files, etc. The `trans` parameter is of type `transform_type` and is equal to `identity`, `rotate_90`, `rotate_180`, `rotate_270`, `reflect_0`, `reflect_45`, `reflect_90` or `reflect_135` corresponding to no transformation, 90°/180°/270° rotation, or reflection in $y = 0/y = x/x = 0/y = -x$, respectively applied to the block before inclusion. The `x` and `y` parameters are integers and give the coordinates of the bottom-left cell of the (transformed) sub-block.

4. Tools for CFG format files

Note: Command qualifiers are introduced by "`-`" symbols (as shown below) under Unix, but must be introduced by "`/`" symbols under MS/DOS.

When a CAL circuit has been created in CFG format, the programming information for the CAL array can be generated using the Algotronix cfg2cal tool. As well as a CFG file, this tool also requires a file containing details of the CAL board on which the circuit is to be implemented. The extra cfg2brd tool can be used to generate the board file for a CHS2x4 board from the CFG file as long as the circuit fits within the CHS2x4 CAL array, and has all of its input and output ports on the perimeter of the CHS2x4 CAL array. The command "`cfg2brd` *name*" reads a CFG file with the name *name* suffixed by "`.cfg`" and generates a board file with the name *name* suffixed by "`.brd`". After this, the command "`cfg2cal` *name*" reads both the CFG file and the board file and generates a programming file with the name *name* suffixed by "`.cal`". Inspection of the CAL descriptions produced by cfg2cal may cause surprise, for two main reasons. First, cfg2cal sometimes inverts cell functions to compensate for inversions along routing wires. Second, it eliminates parts of the circuit which do not lead to any output ports; this can be suppressed by giving the "`-r`" qualifier to the cfg2cal command.

To produce pictures of CAL circuits, the extra cfg2ps tool can be used. It reads a CFG file and produces a file containing a Postscript representation of the circuit (excluding any non-perimeter ports). The command "`cfg2ps` *name*" reads a CFG file with the name *name* suffixed by "`.cfg`" and generates a Postscript file with the name *name* suffixed by "`.ps`". Three qualifiers can be given to the cfg2ps command. First, the qualifier "`-w` l,b,r,t" indicates that (l,b) and (r,t) are the coordinates of the bottom-left and top-right cells to be included (the default is to include the whole block). Second, the qualifier "`-g` wxh" indicates that w and h are the width and height in 1/72 inch units of the picture to be produced (the default is the size of an A4 page minus margins). Finally, the qualifier "`-t`" indicates that the Postscript file should be made suitable for inclusion in TEX documents using the epsf macro package (the default is not to do so).

To analyse the timing of CAL circuits conforming to the design guidelines in Section 2, the extra CANAL tool can be used. It reads a CFG file and attempts to partition the circuit into three 'chunks': (a) cells with no function inputs dependent on input ports or on $G_1/\overline{G_1}$-clocked latches, (b) cells with function inputs dependent on input ports or on G_1-clocked latches and (c) cells with function inputs dependent on $\overline{G_1}$-clocked latches. Then, a table is produced for each chunk, showing the earliest time at which each of its cells can produce a stable function output, relative to the time at which (a) the circuit is initialised, (b) G_1 goes high, and (c) G_1 goes low, respectively. This enables critical paths to be identified, as well as minimum durations for G_1 and $\overline{G_1}$ when the circuit is in operation. The command "`canal` *name*" reads a CFG file with the name *name* suffixed by "`.cfg`" and outputs the timing tables to the standard output stream.

5. The CRUN library

CAL circuits can be exercised on the CHS2x4 board using the graphical DEBBIE tool or the program-based LIB2x4 C library, both from Algotronix. The extra CRUN library is an alternative program-based C library which is simpler than LIB2x4, because it hides most of the details of the CHS2x4 board. To configure the CAL array, the program calls:

```
configure_cal (name)
```

where the `name` parameter is a string which is the name of a file containing the CAL programming information. A board configuration file named "`chs2x4.cnf`" must also exist in the current directory. Note that if the CAL circuit does not occupy all of the available CAL1024 chips, then the spare chips will not be configured and so may retain undesirable circuitry from an earlier configuration. To initialise the CHS2x4 control circuitry to a known state, the program calls:

```
reset_cal ().
```

After this, if the circuit conforms to the timing guidelines in Section 2, it can be exercised by the program driving it through a sequence of operational steps by calling:

```
run_cal (new_g2, steps, source, dest, mon, monproc)
```

where the `new_g2` parameter is of type Bool and is equal to TRUE if the global signal G_2 is to be high during the computational phase of each step, or FALSE if G_2 is to be low during the computational phase of each step. The `steps` parameter is an integer (≥ 1) which indicates the number of operational steps to be performed. The `source` parameter is a pointer to an array of `steps` bytes (unsigned chars) which are to be supplied in order as inputs to the circuit, one per step. The `dest` parameter is a pointer to an array of `steps` bytes (unsigned chars) which are to receive in order the outputs of the circuit, one per step. If desired, it is safe for both `source` and `dest` to point to the same array. The `mon` parameter is of type Bool and the `monproc` parameter is a function with a single integer parameter and no result; `mon` is equal to TRUE if `monproc` is to be called during the computation phase of each step with a parameter equal to the step number (in the range $0, \ldots, \texttt{steps} - 1$), or FALSE if not. This arrangement can be used to output diagnostic information, or even to artificially extend the duration of the computation phase.

6. Availability

The non-Algotronix software described in this document was written to support the Bit Stream Enhancement (BSE) project of the author. It is under continuous development, and is unsupported for any other applications. Prototype versions of the software may be made available by the author to others on an 'as is' basis, but only for use in non-profit making organisations and only for non-military related applications.

Appendix. Simple design example — a five-stage shift register

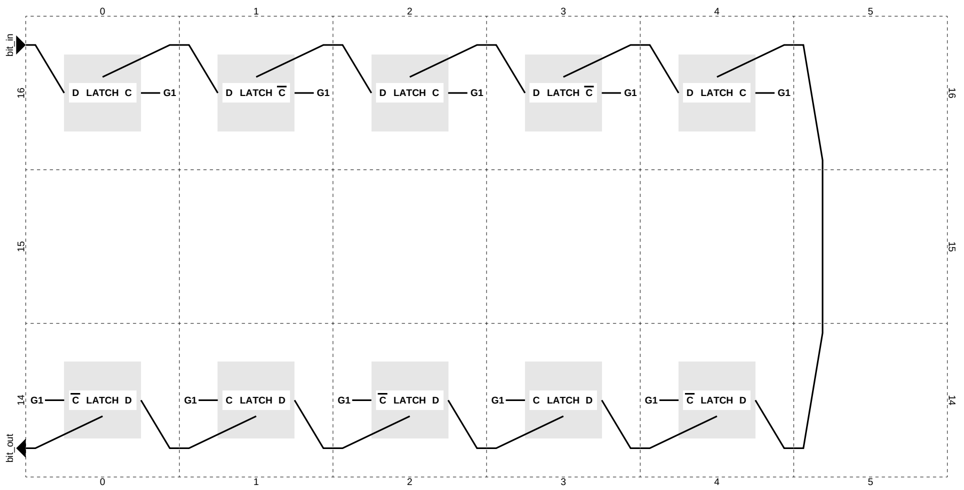
(a) C program using CLAP to create the circuit in a CFG file

```
#include "clap.h"

main (argc, argv)
int argc;
char *argv [];
{
   int x;

   begin_block ("shift");
      block_port ("bit_in", in_port, 0, 16, left);
      for (x = 0; x <= 4; x++) {
         begin_cell (x % 2 == 0 ? c_d_latch : not_c_d_latch, x, 16);
            from_to (g1, c_in);
            from_to (left, d_in);
            from_to (f_out, right);
         end_cell ();
      }
      wire_cell (left, bottom, 5, 16);
      wire_cell (top, bottom, 5, 15);
      wire_cell (top, left, 5, 14);
      for (x = 4; x >= 0; x--) {
         begin_cell (x % 2 == 0 ? not_c_d_latch : c_d_latch, x, 14);
            from_to (g1, c_in);
            from_to (right, d_in);
            from_to (f_out, left);
         end_cell ();
      }
      block_port ("bit_out", out_port, 0, 14, left);
   end_block ();
}
```

(b) Picture of circuit produced by cfg2ps from CFG file



(c) Timing tables for circuit produced by CANAL from CFG file

```
Cell completion times in G1-on chunk (min duration = 18 ns)

     |     0     1     2     3     4     5
-----+------------------------------------
  16 |    18          18          18
  15 |
  14 |          18          18


Cell completion times in G1-off chunk (min duration = 18 ns)

     |     0     1     2     3     4     5
-----+------------------------------------
  16 |          18          18
  15 |
  14 |    18          18          18
```

(d) C program using CRUN to exercise the circuit

```
#include "crun.h"

main (argc, argv)
int argc;
char *argv [];
{
   unsigned char in_buff  [12] = { 0,1,1,0,1,1,1,0,0,1,1,0 };
   unsigned char out_buff [12] = { 0,0,0,0,0,0,0,0,0,0,0,0 };
   int i;

   /* Configure using file produced by cfg2brd, then cfg2cal */
   configure_cal ("shift.cal");
   reset_cal ();

   /* Drive circuit through 12 operational steps */
   run_cal (FALSE, 12, in_buff, out_buff, FALSE /*, null*/);

   printf ("Inputs to bit_in     :           ");
   for (i = 0; i < 12; i++) printf ("%2d", in_buff [i]);
   printf ("\nOutputs from bit_out : ");
   for (i = 0; i < 12; i++) printf ("%2d", out_buff [i] >> 7);
   printf ("\n");
}
```

(e) Output from above program when run on PC with CHS2x4 board

```
Programming chip (0,0) : Address=0

Inputs to bit_in     :           0 1 1 0 1 1 1 0 0 1 1 0
Outputs from bit_out :  1 0 1 0 0 0 1 1 0 1 1 1
```

Note: the first five bits output are the 'random' initial contents of the shift register.