# CamelWare

Ocaml, bit by bit

Documentation of the `CamelWare` language

## Overview

`CamelWare` is a functional hardware description language for digital logic. It provides syntactical constructs that make it easy to compactly describe the construction of small to medium scale digital circuits and an interactive simulator for examining their behavior.

### Keywords

1. `input`
2. `output`
3. `register`
4. `falling`
5. `rising`
6. `fun`
7. `let ... in ...`
8. `if ... then ... else ...`

## Structure of a `CamelWare` program

A `CamelWare` program consists of a set of component definitions. These fall into four categories:

- Registers
- Inputs
- Outputs
- Subcircuits

The order of definitions does not matter to the compiler except in that if multiple definitions are given the same name then only the last one will be used.

## Registers, Inputs, and Outputs

The definition syntax has the same general form for registers, and outputs. It is:

```
definition := type name[length] = expression

type := [rising | falling] register | output
```

Registers and outputs are assigned a transition function (this is the `expression` after the `=`). Whenever it comes time to compute their next value this is evaluated. Rising registers become the result of applying their transition function to the current state (ie they step) when the clock changes from `0` to `1`. Falling registers step when the clock changes from `1` to `0`. Outputs reevaluate their transition function whenever anything else in the circuit changes (This essentially simulates instantaneous logic). Although you can specify `rising` if you wish, registers are assumed to be rising by default - you must add the optional `falling` keyword to their declaration to make them so.

Defining an input is simply:

```
input name[length]
```

## Transition Functions

`CamelWare` models circuits at the register transfer level. That is, circuit definitions don't consist of lists of individual gates - instead they consist of lists of registers and how they relate to each other. Each register is defined with a transition function, which specifies how its next value is computed each time it steps. A transition function can be thought of as implicitly defining the wires and gates neccesary to actually implement the computation in question in hardware. Each transition function is simply a logical expression defined in terms of the registers and inputs in the current state of the circuit

If the result of a transition function is longer than the length of the register to which it is to be assigned then it is implicitly truncated. If it is shorter it is zero extended.

## Constants

### Syntax

`CamelWare` allows referring to constants as hexadecimal, binary, and decimal numbers. The syntax for these is:

```
constant := binary | hexadecimal | decimal
binary := [length]`b bin_digit+
hexadecimal := [length]`x hex_digit+
decimal := [length]`d [-] dec_digit+
```

If the length is left unspecifed then the maximum possible length (32 bits) will be assumed. If the length is larger than the number of digits actually specified then the remaining digits will be assumed to be zero. If it is smaller then the most significant digits will be cut off to make it fit. Only decimal constants are allowed to be explicitly negative. Binary and hexadecimal constants correspond more directly to the bits and can be made implicitly negative by specifying their MSB to be 1.

**Example Usage** *[length]'[base][value] where base is in b for binary, d for decimal, x for hexidecimal and 0 < length <= 32*

```
2'b10
32'b00000000000000000000000111010111000
32'd10
10'd1
4'x4
32'x314AFF0E
5'd-4
```

## Manipulating Values

### Array Access

Referring to the current value of registers and inputs is easy - you simply use their name in your transition function. It is illegal to refer to the current value of an output. Since each register consists of an array of bits, it is frequently desirable to access a specific subsequence. The syntax for this is:

```
nth_bit := expression[index]
subsequence := expression[index1 - index2]
```

Accessing subsequence `A[from - to]` gives bits `from` through `to` inclusive of `A`. All values are zero indexed with the indices in order of increasing significance.

### Concatenation

It is also possible to concatenate values together. We do this by surrounding a comma delineated list of expressions we wish to concatenate with the brace symbols `{` and `}`.

```
concatenate := {expression1, expression2, ...}
```

The first expression in the list becomes the most significant subsequence of the resulting value.

### Example Usage

```
(5'b00010)[0] -->* 1'b0
(5'b00010)[1] -->* 1'b1
(5'b00010)[2] -->* 1'b0
(5'b00010)[3] -->* 1'b0
(5'b00010)[4] -->* 1'b0
(5'b00010)[0-2] -->* 3'b010
(5'b00010)[1-3] -->* 3'b001
{2'b11, 3'b000} -->* 5'b00011
```

## Operators

### Basic Hardware Gates

`CamelWare` supports a number of operators which correspond to basic hardware gates. These are:

1. `~` bitwise not
2. `&` bitwise and
3. `|` bitwise or
4. `^` bitwise xor
5. `~&` bitwise nand
6. `~|` bitwise nor
7. `~^` bitwise nxor

These can be applied as follows:

```
bitwise_operation := expression1 bitwise_operator expression2
bitwise_operator := & | | | ^ | ~& | ~| | ~^
bitwise_negation := ~expression
```

When the two operands of a gate are not of equal length, the shorter of the two is zero extended to reach the length of the other. The output of the expression is of length `max (length expression1) (length expression2)`.

### Example Usage

```
3'b001 | 3'b010 -->* 3'b011
3'b001 & 3'b010 -->* 3'b000
~3'b001 -->* 3'b110
```

### Reduction

`CamelWare` supports a `Verilog` inspired "reduction" syntax. This allows all of the bits of an expression to be reduced to a single value using a gate.

```
reduction := bitwise_operator expression
```

For example `|(3'b001)` is equivalent to `1'b0 | 1'b0 | 1'b1` which evaluates to `1'b1`. `&(1'b001)` on the other hand would evaluate to `1'b0`.

### Example Usage

```
^(3'b111) -->* 1'b1
&(3'b101) --->* 1'b0
```

### Logical Operators

Traditionally, logical operators obey the following semantics: all non zero values are treated as `true`, and zero is treated as `false`. `CamelWare` supports this by means of following operators:

1. `!` logical not
2. `&&` logical and
3. `||` logical or

Logical operations obey the same length extension rules as bitwise operations on inputs of different sizes. The output of a logical operation is either `1'b0` or `1'b1` depending on the result.

### Example Usage

```
!5'b10011 -->* 1'b0
!5'b00000 -->* 1'b1
5'b10011 && 5'b11000 -->* 1'b1
5'b00000 && 5'b11000 -->* 1'b0
```

### Arithmetic Operators

`CamelWare` supports twos complement arithmetic operations. These are:

1. `-` arithmetic negation

2. `+` addition
3. `-` subtraction

When the two operands of an arithmetic operator are not of the same length the shorter of the two is sign extended. Just as with bitwise operations, the result is of length `max (length expression1) (length expression2)`.

### Example Usage

```
32'd1 + 32'd2 -->* 32'd3
-3'b001 -->* 3'b111
```

### Relations

To compare twos complement values we use the following operators

1. `<` less than
2. `<=` less than or equals
3. `>` greater than
4. `>=` greater than or equals
5. `==` equals
6. `!=` not equals

Just as with arithmetic, before comparison the two operands are sign extended to match. The output is of length 1.

## If Then Else

### Syntax

To conditionally evaluate an expression the appropriate syntax is:

```
if_then_else := if condition then expression1 else expression2
```

In hardware terms this is equivalent to a 2 to 1 multiplexer with `condition` as the selector bit. To support the logical semantics where all nonzero values are treated as true, if `condition` is longer than 1 bit then it is first reduced with `|` before being fed into the mux. The output of an `if then else` is of length `max (length expression1) (length expression2)` with the shorter of the two expressions zero extended if neccessary.

### Example Usage

```
if 1'b0 then 3'b010 else 3'b001 -->* 3'b001
if 1'b1 then 3'b010 else 3'b001 -->* 3'b010
```

## Let Expressions

### Syntax

To avoid lengthy and cumbersome repeated subexpressions, `CamelWare` supports an `OCaml` like `let ... in` syntax. This is:

```
let variable = expression1 in expression2
```

In `expression2`, `variable` is bound to the result of evaluating `expression1` and can be referenced by name.

### Example Usage

```
let x = 3'b010 in x -->* 3'b010
let x = 3'b010 in 1'b1 -->* 1'b1
```

## Subcircuit Definitions

### Syntax

In order to allow for code reuse `CamelWare` allows for defining subcircuits. These are essentially functions with multiple inputs but only one output. The syntax is:

```
subcircuit := fun function_name (arg1[length1], ...)[output_length] = expression
```

To apply it:

```
application := function_name(expression1, ...)
```

To ensure that indexing isn't out of bounds, input expressions are all zero extended or truncated to become the length specified in the function definition before it is applied. The function's result is similarly adjusted to become `output_length`.

A function corresponds to a subcircuit in hardware. When function `f` calls function `g` it means that subcircuit `f` contains `g` physically. Accordingly any sort of recursion is completely illegal.

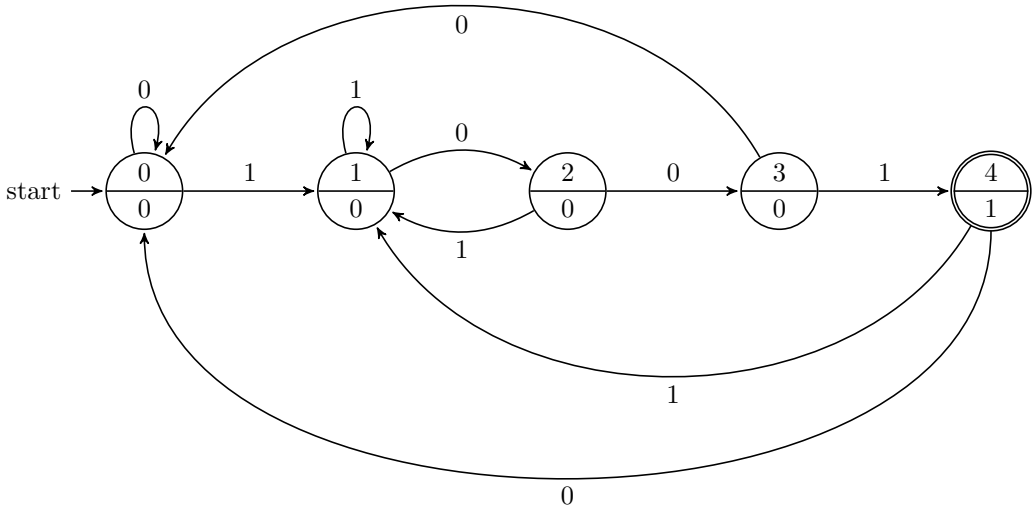
### Example Usage

```
fun f(x[1],y[1])[1] = x & y
fun g(x[4])[2] = {g[0],g[3]}
```

## Large Examples

### State Machine

The following state machine processes an incoming string of bits via an input. It outputs `1'b1` when it receives the string `1001`




A naive `CamelWare` implementation is:

```
input in_channel[1]
register state[3] = next(in_channel, state)
output out_channel[1] = state == 3'b100
fun next(i[1],s[3])[3] =
  if s == 3'd0 then
    if i
    then 3'd1
    else 3'd0
  else if s == 3'd1 then
    if i
    then 3'd1
```

```
  else 3'd2
else if s == 3'd2 then
  if i
  then 3'd1
  else 3'd3
else if s == 3'd3 then
  if i
  then 3'd4
  else 3'd0
else
  if i
  then 3'd1
  else 3'd0
```

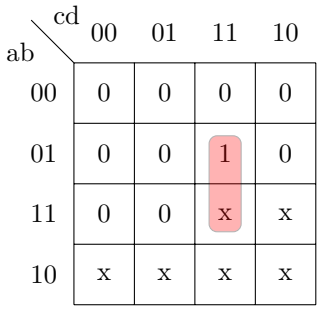
Although the above code is very readable, unfortunately it suffers from serious efficiency problems. `CamelWare` does not have an optimizing compiler - there's a one to one correspondence between operators and gates - and this means that the above uses 5 comparators and 9 2 to 1 multiplexers to acheive some fairly simple functionality.

A better approach is to use traditional digital design methods to work out the logic before writing it down. The desired truth table is:

| State[2] | State[1] | State[0] | In_channel | Next | Out_channel |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 3'b000 | 1'b0 |
| 0 | 0 | 0 | 1 | 3'b001 | 1'b0 |
| 0 | 0 | 1 | 0 | 3'b010 | 1'b0 |
| 0 | 0 | 1 | 1 | 3'b001 | 1'b0 |
| 0 | 1 | 0 | 0 | 3'b011 | 1'b0 |
| 0 | 1 | 0 | 1 | 3'b001 | 1'b0 |
| 0 | 1 | 1 | 0 | 3'b000 | 1'b0 |
| 0 | 1 | 1 | 1 | 3'b100 | 1'b0 |
| 1 | 0 | 0 | 0 | 3'b000 | 1'b1 |
| 1 | 0 | 0 | 1 | 3'b001 | 1'b1 |

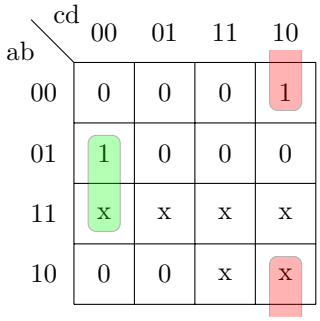
The Karnaugh maps are

`Next[2]`:



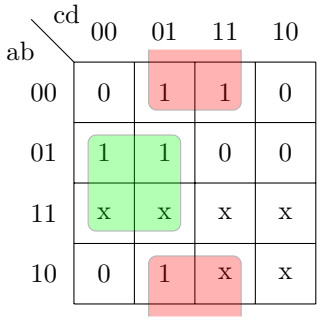

`Next[2] = State[1] & State[0] & In_channel`

`Next[1]`:




`Next[1] = ~State[1] & State[0] & ~In_channel | State[1] & ~State[0] & ~In_channel`

`Next[0]`:



```
Next[0] = ~State[1] & In_channel | State[1] & ~State[0]
```

`Out_channel`:

| ab \ cd | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 0 |
| 01 | 0 | 0 | 0 | 0 |
| 11 | x | x | x | x |
| 10 | 1 | 1 | x | x |

```
Out_channel = State[2]
```

From this we can derive an efficient hardware implementation

```
input in_channel[1]
register state[3] = next(in_channel,state)
output out_channel[1] = state[2]
fun next(i[1],s[3])[3] = {
  s[1] & s[0] & i,
  ~s[1] & s[0] & ~i | s[1] & ~s[0] & ~i,
  ~s[1] & i | s[1] & ~s[0]
}
```

### Shift Registers and Counters

The following examples are inspired by the the example designs in section 4.3 of **FPGA Prototyping By Verilog Examples** by Pong P. Chu.

**Free Running Shift Register** A free running shift register shifts in the input bit on the right and shifts out the output bit on the left each clock cycle. The `CamelWare` code for this is

```
input in_channel[1]
register R[32] = {in_channel,R[1-31]}
output out_channel[1] = R
```

**Universal Shift Register** A universal shift register either shifts to the left, shifts to the right, loads parallel data, or stays the same depending on a 2 bit control signal. It outputs the current contents of the register.

| Ctrl[1] | Ctrl[0] | Operation |
|---|---|---|
| 0 | 0 | Pause |
| 0 | 1 | Shift Left |
| 1 | 0 | Shift Right |
| 1 | 1 | Load |

```
input in_channel[32]
input ctrl[2]
register R[32] = next(in_channel,ctrl,R)
output out_channel[32] = R
fun next(i[32],c[2],r[32])[32] =
  if c == 2'b00 then r
  else if c == 2'b01 then {r[0-30],i[0]}
  else if c == 2'b10 then {i[31],r[1-31]}
  else i
```

**Free Running Binary Counter** A binary counter increments its value by one each tick until eventually overflowing and returning to `0`. The following is a 8 bit example:

```
output max_tick[1] = C == 8'b11111111
register C[8] = C + 8'd1
```

**Universal Binary Counter** A universal binary counter counts up, counts down, pauses, is loaded with a new value or is synchronously cleared depending on the value of a 4 bit control signal.

| Clear | Load | Enable | Up | Next | Operation |
|---|---|---|---|---|---|
| 1 | x | x | x | 32'x00000000 | Clear |
| 0 | 1 | x | x | In_channel | Load |
| 0 | 0 | 1 | 1 | C + 1 | Up |
| 0 | 0 | 1 | 0 | C - 1 | Down |
| 0 | 0 | 0 | x | C | Pause |

```
input in_channel[32]
input ctrl[4]
register C[32] = next(in_channel,ctrl,C)
output max_tick[1] = C == 32'xFFFFFFFF
fun next(i[32],c[4],counter[32])[32] =
  if c[3] then 32'x00000000
  else if c[2] then i
  else if c[1] then
    counter + (if c[0] then 32'd1 else 32'd-1)
  else counter
```