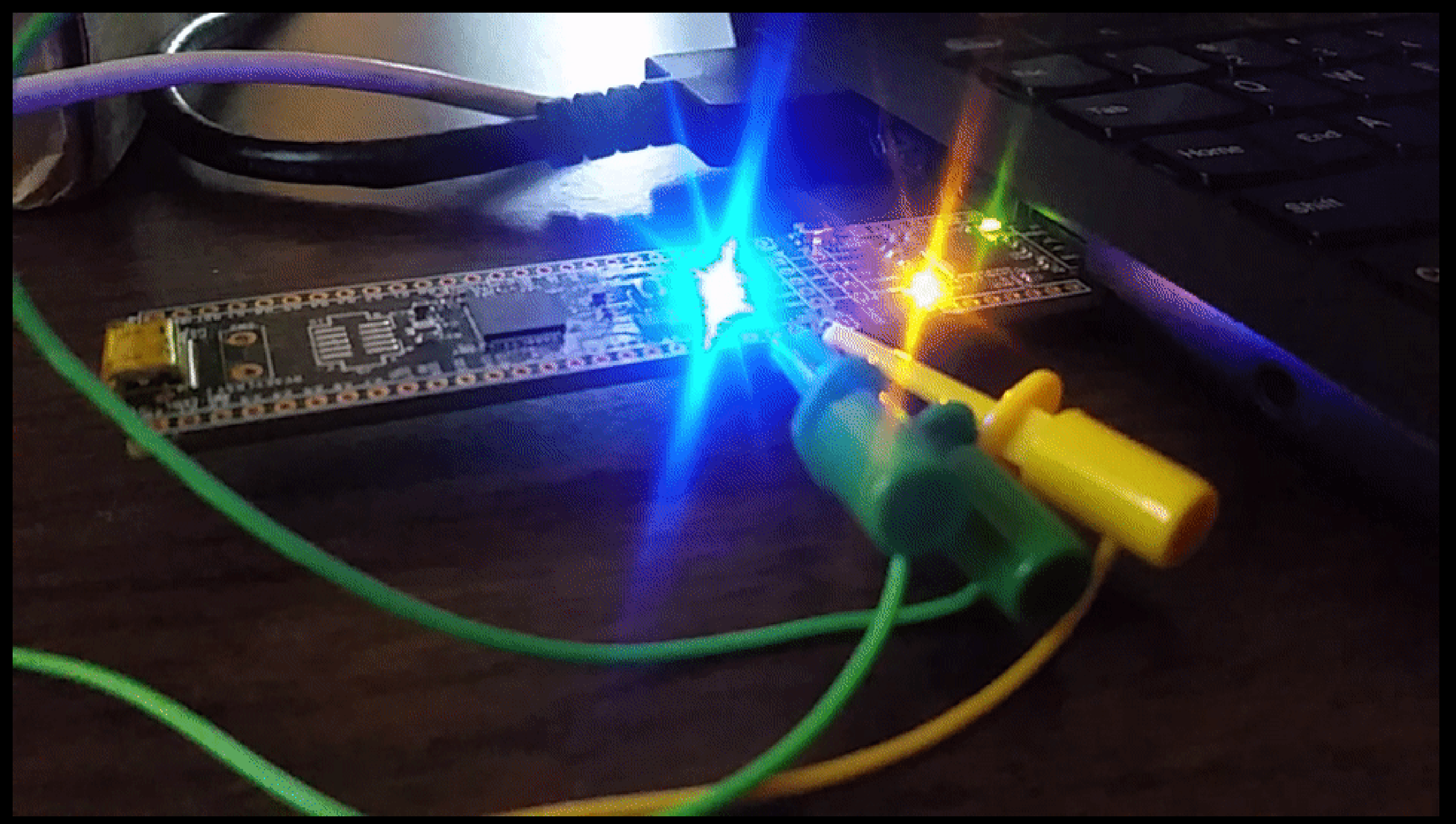
gelFORTH | A Forth for interactive hardware design.



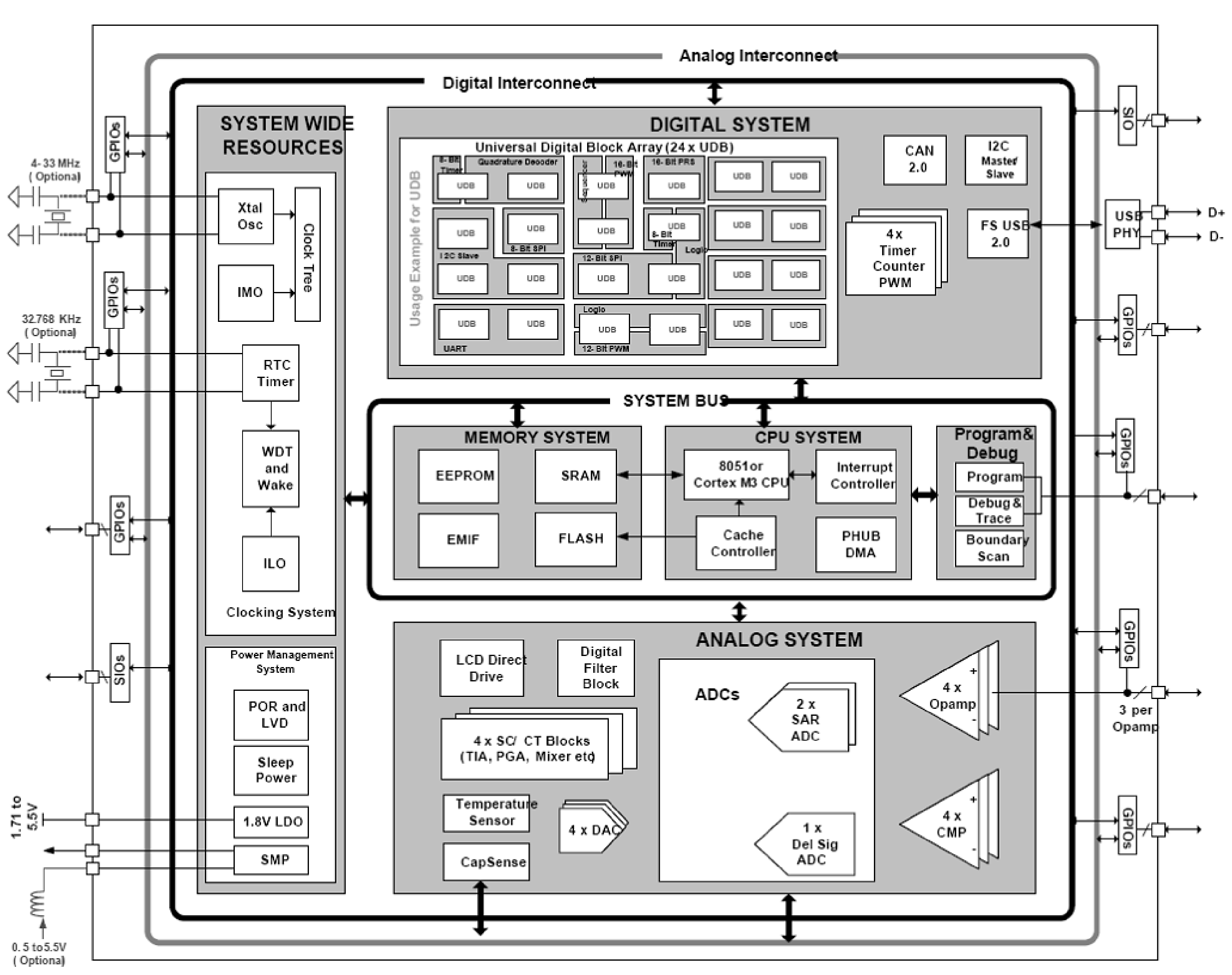
Andreas Wagner
andreas.wagner@lowfatcomputing.org
http://www.0xFF.in
@lowfatcomputing

The Cypress PSoC 5LP (What is it?)

an SoC with:

- an ARM Cortex M3
- a Universal Digital Block “UDB” array
  - reconfigurable logic & routing
  - something like a CPLD
- a Universal Analog Block “UAB” array
  - reconfigurable logic & routing
- Proprietary Windows Only Toolchain :(
- Routing fabric is undocumented :(

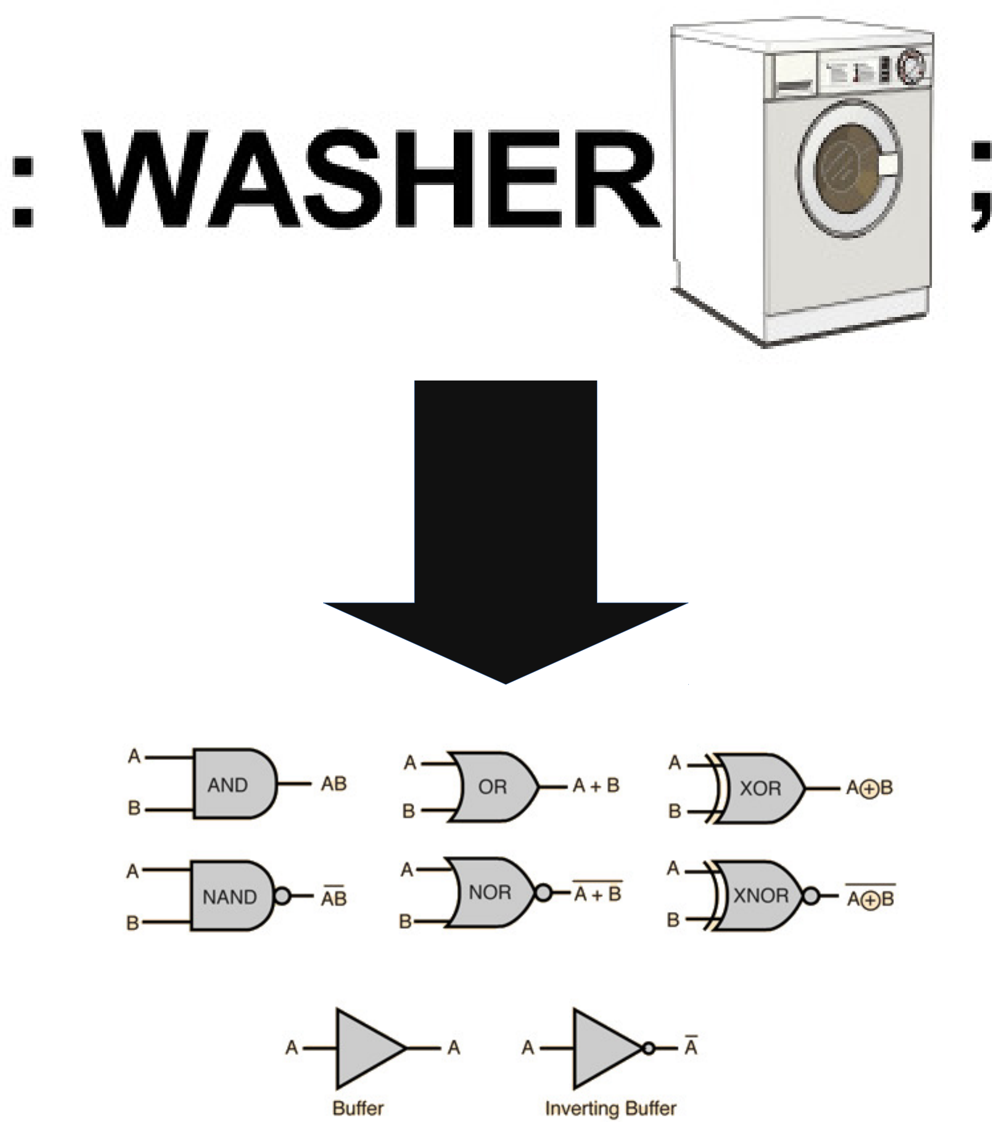
The Cypress PsoC 5LP (block diagram)



Interactive Hardware Design. Why?

- Electronics multi-tool with loadable “hardware” modules and good performance for them.
- Free/Open Source. No MS Windows IDE necessary.
- Rapidly and interactively explore hardware designs with the same code used to write software.

Forth → Gates
(has anyone done it?)



Virtual Hardware Definition Language

Using Forth as a VHDL

Abstract

A set of VHDL extensions to Forth lets programmers define hardware in the same language with which they write software. Hardware defined in Forth can be verified by executing the hardware-definition words at the command line or by writing special Forth words to test their operation. The use of the same language for hardware and software simplifies the task of swapping hardware and software functions during optimization.

Introduction

Computer-aided design has become an essential part of product development, and several different hardware definition languages (HDLs) are marketed for that purpose, but I wanted to define the hardware in the same language the software was written in. We have been using Forth to define PAL equations for about ten years using a set of extensions to Forth called CARMAP. When we started using complex programmable logic devices (CPLDs), it seemed more logical to extend CARMAP than to buy an off-the-shelf compiler and learn a new language.

With each improvement, CARMAP has moved closer to

Designing logic with the Forth VHDL

1. Write a software simulation of the design.
2. Test the design.
3. Convert the software simulation into a hardware definition.
4. Compile the hardware definition into logic equations.
5. Fit the logic equations into the device.
6. Verify that the logic equations work correctly.
7. Route the signals and assign the I/O pins.
8. Convert the routed design into a fuse map.

Simulation

The simulation of a design allows interactive analysis of many aspects of the hardware including complexity, functionality, timing, and performance. If the application program for the proposed hardware is also written in the same language as the hardware, hard and soft components can be interchanged during optimization.

The software model

The software model is like a black box, it doesn't matter how it works as long as it works correctly. The main advan-

Page: 22 of 80 100%

number of terms may be possible.

How the logic compiler works (CARMAP)

The logic compiler converts each of the functions described in Forth into a set of logic equations for each output bit of the function. This is a conceptually simple process that involves expanding the function into a truth table and then reducing the number of terms in the truth table to the minimum.

The function is mapped into the truth table using the inputs that are related to the output. After the function has been mapped, the table is scanned for unused inputs. If any unused inputs are found, they are removed from the table. Each input that is removed cuts the table size by half.

The truth table is then converted into logic equations by an exhaustive scanning process that tries all possible combinations of inputs and compares them with the truth table. The first step is to search the table for a true output. When an output is found, all sets in which it resides are tested for correlation with the other outputs. The largest true set is saved, and the bits within it are marked as solved. Then the next unsolved output is found and the process repeats until finished.

The second step of the transformation is to delete the sets in which all elements have more than one solved mark. This gives something close to the ideal two-level array. Fitting the logic into a FPGA would require a third step to convert this ideal array into a multi-level array that would fit into their finer structure. This could be accomplished by recursively factoring gates from the high-level sets and ORing them together.

Conclusion

Forth provides a good foundation for a VHDL system because Forth is an extensible virtual interpreter. Most everyone who works with Forth knows its unique features can enhance software productivity. My experience has shown it to be very useful when working with variable hardware, as well.

The Forth inner interpreter is a very simple list processor that requires only three pointers, two registers, and an ALU to

Logic."
Clive "Max" Maxfield, *EDN*, 3/1/1996.

Appendix A. CARMAP Word Set

Variables: (Items)

```
MAX:GLB:INPUTS
```

I/O Definitions:

```
IO-GROUP "name"

INPUT "name" [ START BITS ]

OUTPUT "name" [ START BITS CLOCK XORS TERMS
               FLIP USES USEX SEL SELX]
  [ OE PTCLOCK are Lattice-specific commands ]
```

`BITS ( n — )`
A word that defines the number of bits used in an `INPUT` or `OUTPUT`.

`START ( n — )`
A word used in conjunction with `BITS` that sets the starting bit number. If `START` is not specified, the first bit number will be zero.

`CLOCK "name"`
A word that defines the clock for registered outputs.

`XORS ( n — )`
A word that sets the maximum number of inputs to be tried in the XOR term.

`TERMS ( n — )`
A word that sets the maximum number of inputs to a logic block.

`FLIP ( m — )`
A mask that defines which output bits in the truth table will be inverted.

`USES ( m — ) "name"`
A bit mask that defines what bits are used by an output. A counter is a function where each output bit depends on all of the bits less than it. The `USES` mask is rotated to the position of the current output bit. The upper bits in the mask are rotated into the lower bits so they will be used in counting functions.

`USEX ( m — ) "name"`
A bit mask that defines what bits should be tried in an `XOR` function. This word is used in conjunction with `USES`, and the mask rotates the same as for `USES`.

`SEL ( m — ) "name"`
A bit mask that defines a set of bits in a fixed position that are used as a selector. This word is like `USES` but the mask does not rotate .

`SELX ( m — ) "name"`
A bit mask that defines what bits should be tried in an `XOR` function. The mask stays in a fixed position. This word is used in conjunction with `SEL`.

`OE ( — )`
A word that defines an output-enable term for a Lattice device.

`PTCLOCK ( — )`
A word that defines a clock term for a Lattice device.

`END-IO-GROUP`
A word that closes the I/O group.

Software Simulation Words

`>> ( io — ) "label"`
The top element on the stack is moved to the input and output registers. (This word is used for design verification.)

`>>O ( o — ) "label"`
The top element on the stack is moved to the output register.

`>>X ( x — ) "label"`
The top element on the stack is moved to the don't-care register.

`>>OX ( d x — ) "label"`
The top element on the stack is moved to the don't-care register, and the next element is moved to the output register.

`O>> ( — o ) "label"`
The output register is copied to the stack.

`TRUTH-TABLE: ( io-group_ads — ) "simulation_word"`
Builds the truth table for a function, and solves the logic equations.

`MAKEMACS`
Solves all of the logic equations in a design.

Hardware Simulation Words

`INIT-LOGIC`
Must be done before defining nodes.

`NODE "name"`
Creates a single-bit, self-fetching variable called `%name`.

`NODES ( s n — ) "name"`
Creates a multiple-bit, self-fetching variable called `%name`.

`CLOCK "name"`

A word that closes the I/O group.

Software Simulation Words

`INVERT ( d — d )`
The logical `NOT` of the bits in a word.

`MAP[ ( v — ) n`
A word that creates an associative memory structure similar to a `CASE` statement.

`MAP ( v a — )`
A word that inserts a token (*v*) and its associated value (*a*) into the `MAP` structure.

`]MAP ( a — )`
A word that inserts the default value (*v*) into the `MAP` structure, and finishes the mapping function.

`]: ( — a )`
A word that changes the state to compilation and returns the address of the start of the compiled string.

`;[`
A word that inserts a `next` into the compiled string and changes the state back to interpret.

Creates a multiple-bit, self-fetching variable called `%name`.

`CLOCK "name"`
Creates a single-bit, self-fetching variable called `%%name`.

`UPDATE-STATE`
Updates the state of the outputs for all functions.

`EXECUTE-CLOCK`
Copies the state of the outputs to the inputs.

`SIMLDF`
The name of the simulation vocabulary.

Lattice-specific words for defining I/O pins

`CLKMAC ( n io-group_ads — ) "name" FORGET "io-group_name"`

`IOMAC ( n io-group_ads — ) "name" FORGET "io-group_name"`

`IMAC ( n io-group_ads — ) "name" FORGET "io-group_name"`

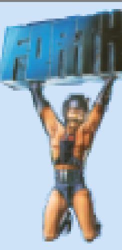
Reconfigurable Architecture Computation Engine

RACE™

Testra's VHDL in Forth

It's a special purpose lexicon…

...Perhaps it need not be so special.



3

[High Stack] Fine-Grain Concurrency (self.Forth)

submitted 2 days ago * (last edited 2 days ago) by transfire [+2]

In my previous post, I suggested that each word would get its own scratch stack. That idea was *questioned mightily*. But here's the fundamental issue that leads to that choice: In "High Stack" concurrency will be a natural part of the language. In fact I am leaning toward concurrency being the default mode of execution, and synchronous execution must be explicitly instructed. This concurrency can occur at the word level. So it is possible for two words to be running at the same time.

Now assume for a moment that the two words are not accessing RAM, they are just calculating, i.e. pure functions. Nonetheless they must have their own private scratch stacks, or they could clobber each other. So I don't see a way around it. So that's where the private scratch stacks come from. And that being the case, I just have to figure how to handle them efficiently.

Now let's go back to the case where there is access to RAM (or IO). That's a more complicated matter and I have been trying to figure out how best to handle it. One idea is the Ownership Model, like that of Rust[1], though I'm not quite sure how that might be applied to Forth. Another is Transactional Memory[2]. Personally I can't quite understand how TM can be efficient, or perfectly safe, but all claim that it is, and there is even talk of building it into hardware in the future. (Also I don't think it applies to IO?)

Beyond that, there is the Erlang-style Actor Model of *no shared memory* -- everything must be explicitly passed. Yet, despite the impressive speed of Erlang with regard to concurrency, I worry it will prove too much overhead for fine-grained concurrency. CSP (like Go's channels) are similar, so I think they might be in the same boat -- nor am I sure how they could be implemented in a transparent manner.

BTW, with regard to the stack, I have (at least I think I have) a good working model using promises/futures and no reference will ever point to anything on the stack (references only point to things in the heap). So I think I am good there.

Hope that explains a few things. This is probably going to be the most difficult nut for me to crack, so I am hoping the community has some good ideas.

Thanks in advance.

17 comments source share save hide give gold spam remove report lock nsfw hide all child comments

all 17 comments subscribe

Parallel Forth
(new wrapper; same GREAT taste!)

[–] **reepca** [+1] **3 points** 1 day ago

Forth seems like a natural fit for SIMD, where you couldn't really imagine performing multiple steps of a computation at the same time (stack serialization and all that), but the tendency to use the stack means that it's easy to imagine performing multiple instances of the same computation at the same time. Are you trying for the former (MIMD/MISD) or the latter (SIMD)? If you're trying to do MISD with a stack, I think you're going to have some difficulties - the same difficulties people trying to do that in hardware to exploit instruction-level parallelism have. To that end, I'd suggest checking up on what the folks working on superscalar stack machines have been up to.

a b + dup c / swap d * *

can be rewritten as ((a + b) / c) * ((a + b) * d)

I imagine it would be quite difficult to easily infer during compilation that a divide and a multiply could be performed simultaneously. Part of it is that statically analyzing the stack effect can be difficult, because conditionals are a thing. Being able to do that for all possible words would, I imagine, be quite difficult. It might be doable if you outlaw variable stack effects, though.

The way I always thought of handling concurrent threads running in a forth system is to just have each thread have its own data stack and return stack. Is that what you mean by "scratch stacks"?

permalink source embed save save-RES spam remove report give gold reply

Verilog is parallel,
Forth HDL as just parallel Forth?!

- Uses an alternate set of primitive words
- Same high level Forth code on top

...but it runs in parallel.

Ladder Logic or String Diagrams

[–] **phalp** 2 points 10 days ago

First of all, make sure the diagrams are showing up in a fixed-width font and that the line-drawing characters are coming out right. Otherwise they're nonsense. Second, it may help to know that the code being diagrammed is `a b + dup c / swap d * *`.

Ok, so here's another version of the diagram. All three are identical up to horizontal spacing. I increased the spacing on this one to hopefully make it easier to read.

```
a───+──╥─────────── ┌────────*──
  b─╜  ╙──────÷────┘ ─────*──╜
          c───╜       d───╜
```

The symbols `a`, `b`, `+`, `c`, `d`, and `*` are the same as in the Forth code. `÷` is simply division (I used this symbol because my first diagram versions used `/` to represent part of a wire). There are two further symbols:

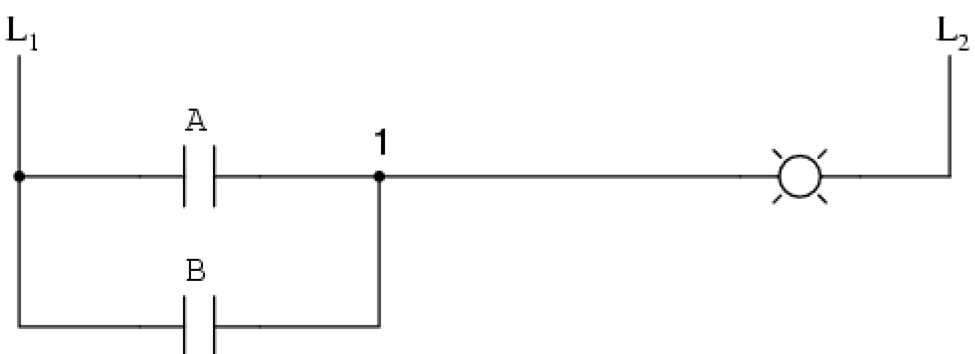
```
 ───╥───     ─── ┌───
    ╙───     ───┘ ───

   dup         swap
```

My concept is that each operator takes a number of items from the stack, represented by strings going into its left side, and leaves a number of items on the stack, represented by strings coming out its right side. Since `dup` takes one items and leaves two, it has one string coming in on the left, and two going out on the right. Since `swap` takes two and leaves two, it has two strings on each side. The math operators all take two and leave one. I used the vertical double line in an attempt to show that the operator is two "lines" high. So if I had an operator that took 3 stack items and left 2, it would appear like one of the following:

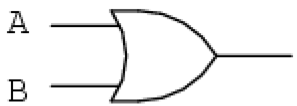
```
───╥───
──?───
───╜
```

When WORDs are independent they are <u>in-parallel</u>:

They are parallel and are leveraging Boolean **OR**

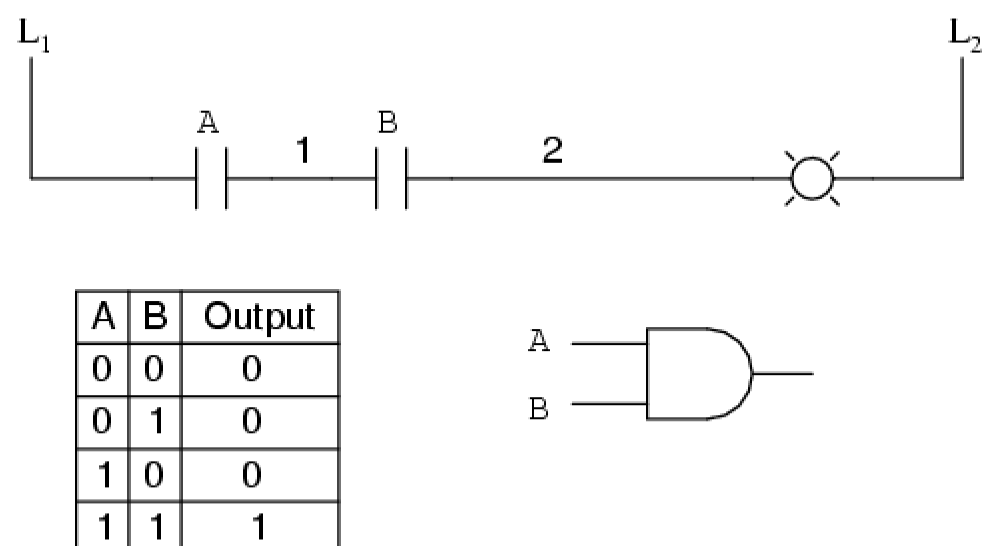


A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1



Where WORDs have dependencies they are <u>in-series</u>:

- The parameters pile up on the stack
- Therefore they must be dealt with in-order.



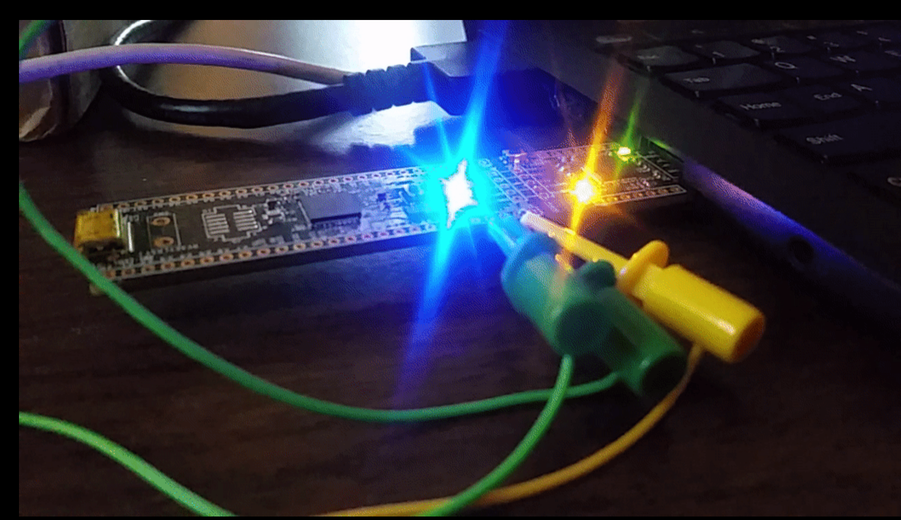
A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

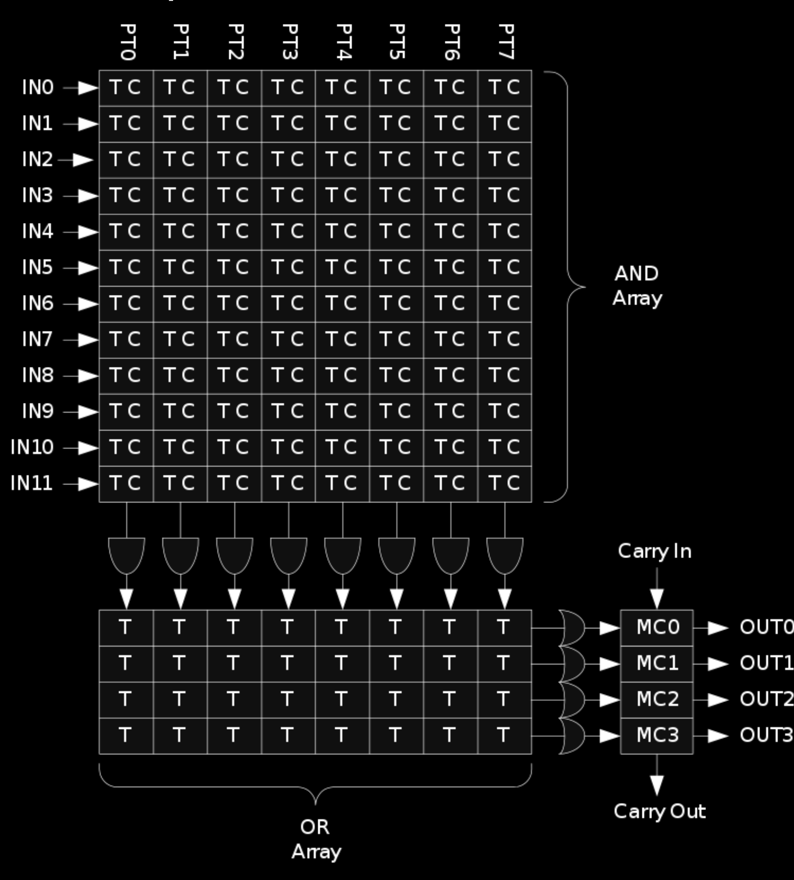
(another reason to minimize stack effects!)

designing the alternate primitive wordset...

an XNOR gate

```
: XNOR
  ( XNOR gate from pins 1.6, 1.7 to blue led at pin 2.1)
  P1.6 @   , P1.7 @   , ( DSI → routing tiles → pi tiles)
  | ( compiles a wire)
  P1.6 @ ~ , P1.7 @ ~ , ( DSI → routing tiles → pi tiles)
  | ( compiles a parallel wire)
  P2.1 ! ( build route to the blue led)
;
```





an XNOR gate

```
: XNOR
   ( route the output of input pins through…)
   P1.6 @   , P1.7 @   , ( DSI → routing tiles → pi tiles)
   |
   P1.6 @ ~ , P1.7 @ ~ , ( DSI → routing tiles → pi tiles)
   |
   P2.1 !
;
```

Port Pins reference VARIABLE data

- Indicate a variable data reference with @
- Compile a variable data reference with **!**

synth alternate wordset: @ !

we can reference a thing…

…but we'll have to build a route to it ourselves.

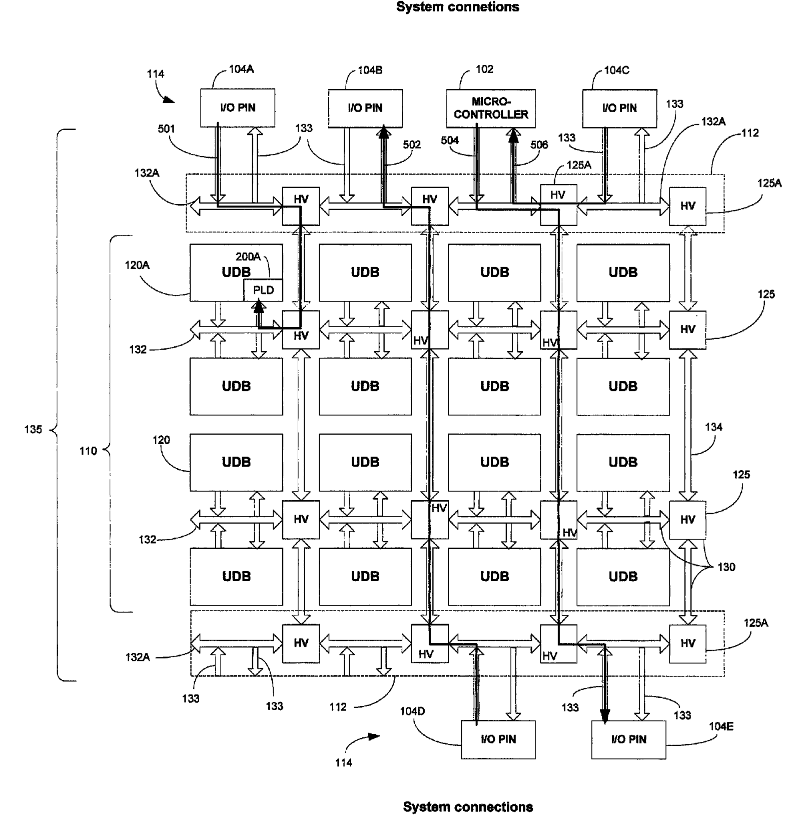


FIG. 11

an XNOR gate

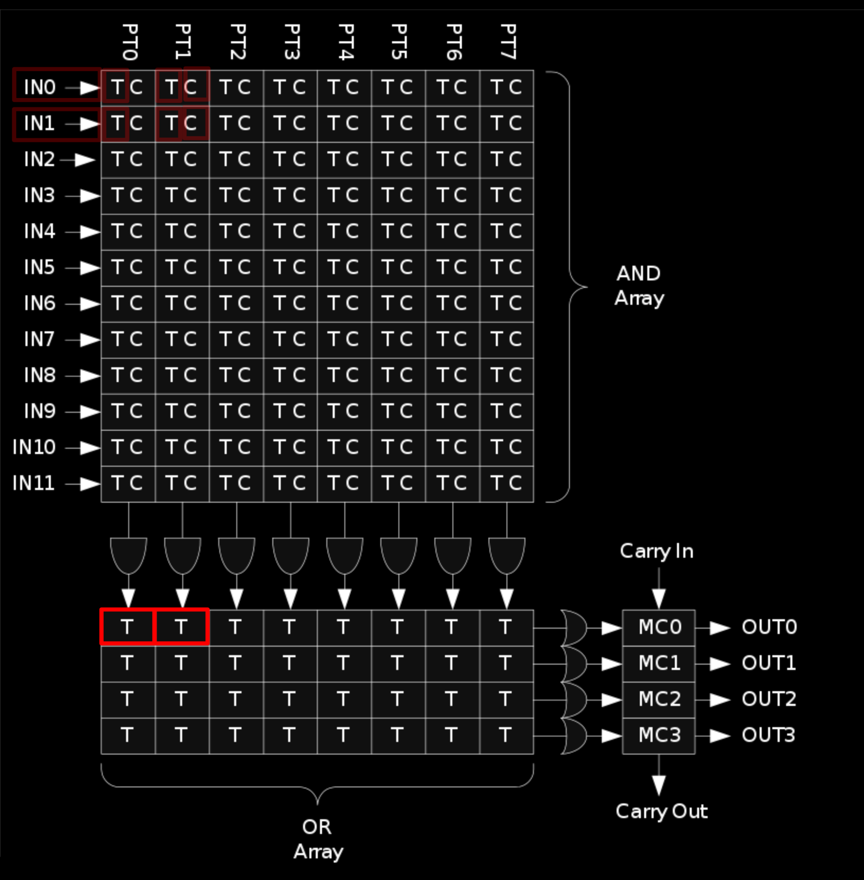
```
: XNOR
   ( compile NOT gates into AND array)
   P1.6 @   , P1.7 @   , ( no complement bits compiled)
   |
   P1.6 @ ~ , P1.7 @ ~ , ( compile complement bits)
   |
   P2.1 !
;
```

an XNOR gate

```
: XNOR
   ( compile YES gates into AND array)
   P1.6 @   , P1.7 @   , ( compile truth bits)
   |
   P1.6 @ ~ , P1.7 @ ~ , ( compile truth bits)
   |
   P2.1 !
;
```

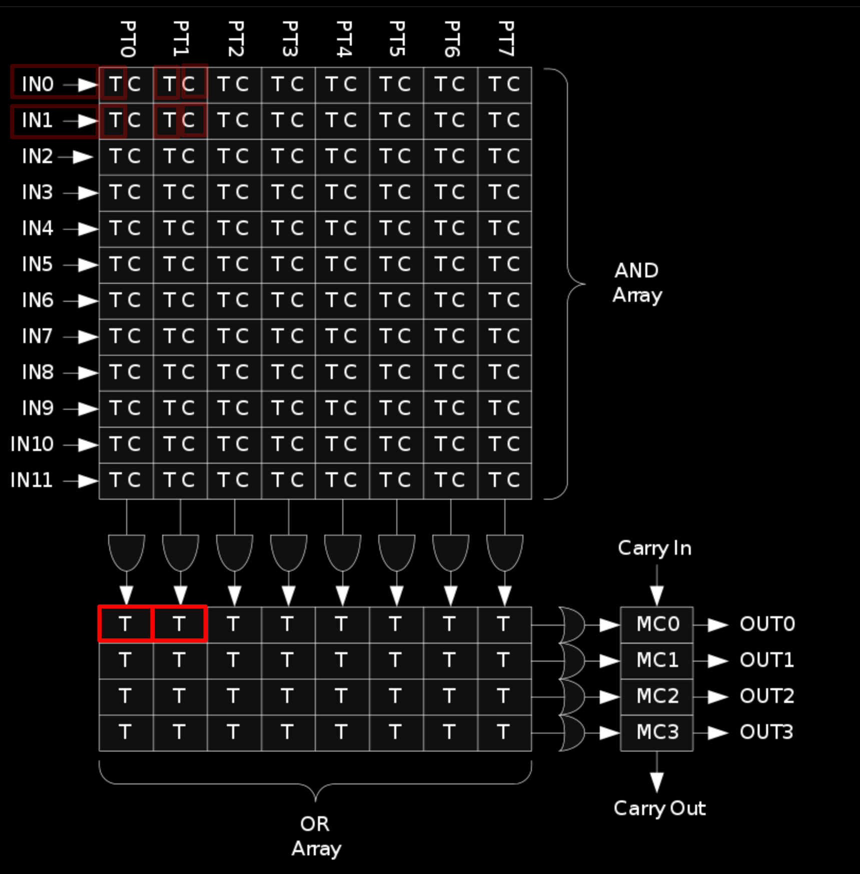
an XNOR gate

```
: XNOR
   ( compile bits to the OR-array)
   P1.6 @   , P1.7 @   ,
   | ( compiles a wire)
   P1.6 @ ~ , P1.7 @ ~ ,
   | ( compiles a parallel wire)
   P2.1 !
;
```



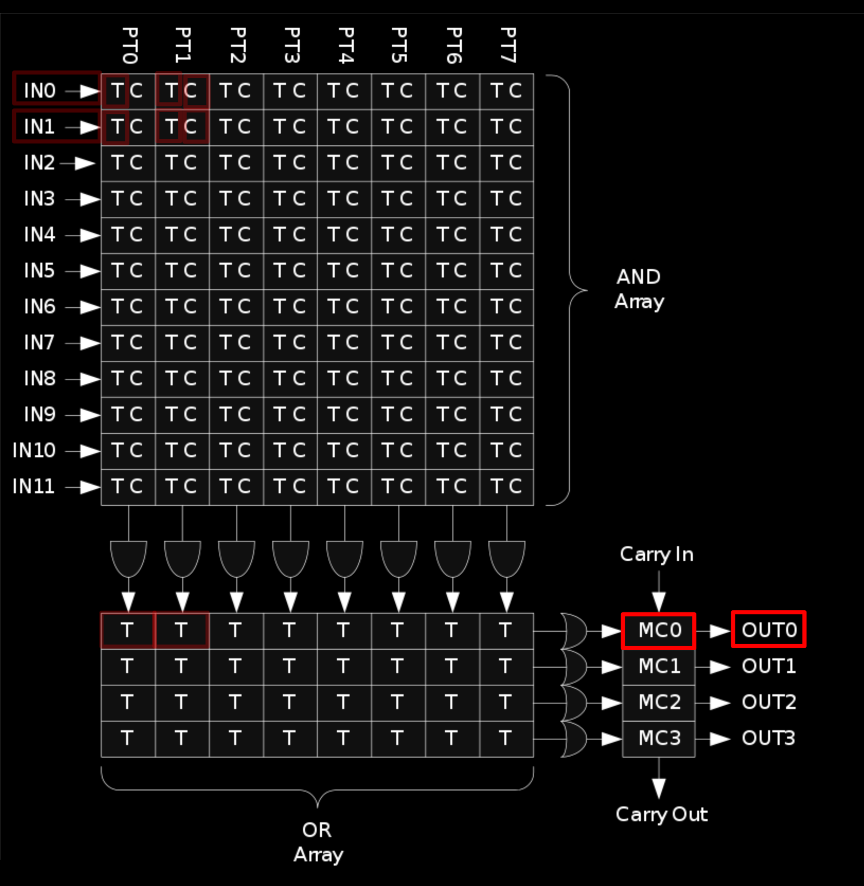
an XNOR gate

```
: XNOR
   ( compile bits to the OR-array)
   P1.6 @   , P1.7 @   ,
   | ( compiles a wire)
   P1.6 @ ~ , P1.7 @ ~ ,
   | ( compiles a parallel wire)
   P2.1 !
;
```

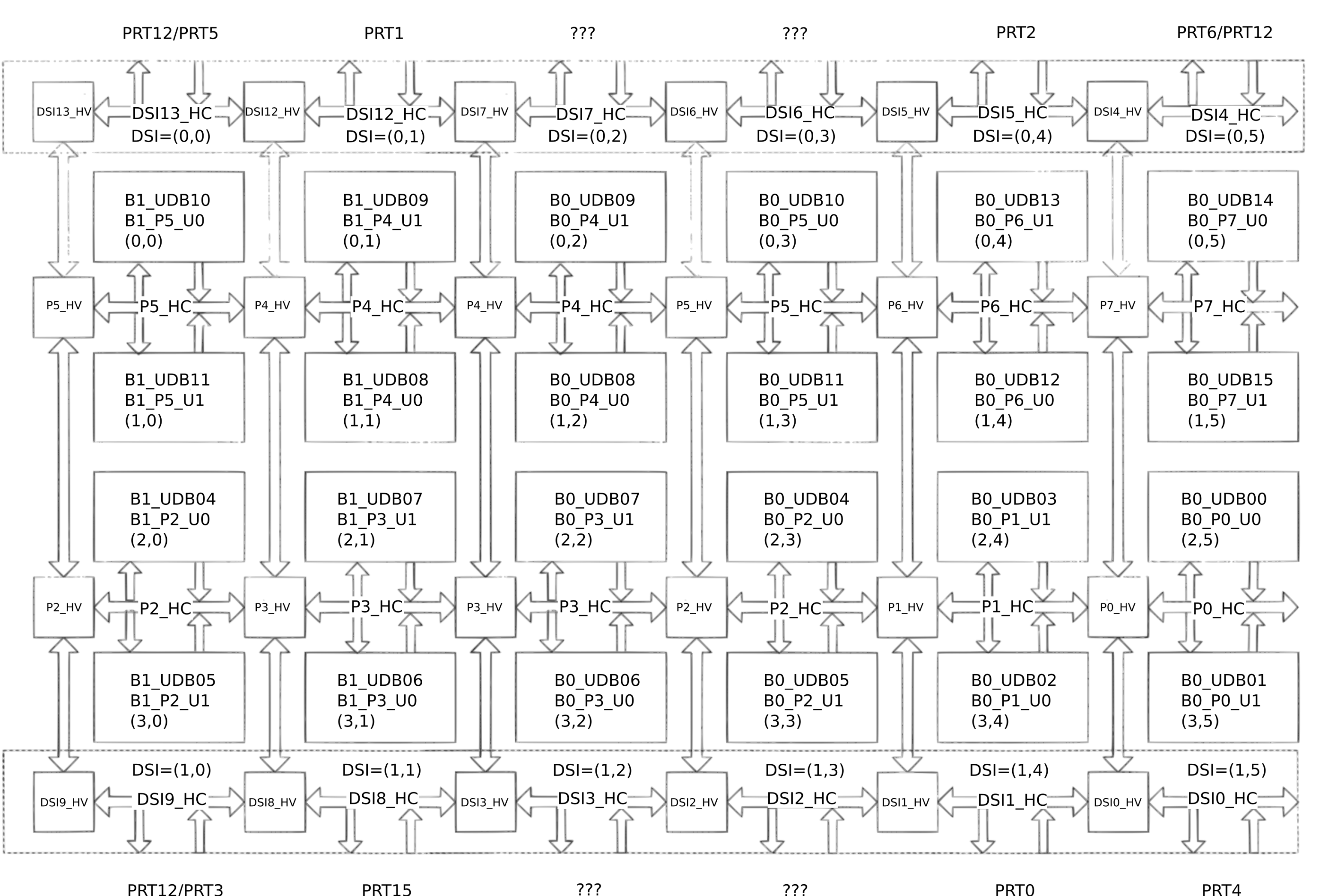


an XNOR gate

```
: XNOR
   ( macrocell and PLD output)
   P1.6 @   , P1.7 @   ,
   | ( compiles a wire)
   P1.6 @ ~ , P1.7 @ ~ ,
   | ( compiles a parallel wire)
   P2.1 !
;
```



synth parallel wordset: ALLOT Maneuvering



Logic Synth words, analogous to regular Forth words

- we're compiling regular Forth to parallel Forth
- synth words have the same names
- …but they compile to the UBD array instead of the Flash or RAM dictionary.

synth alternate wordset: ALLOT Maneuvering

- the addressing scheme is very consistent
- manipulate the addresses to maneuver

ALLOT revolves around YES, NOT, AND, OR

- Logic can spill over into other PLD’s, UDB’s, UDB Pairs, and UDB Banks.
- Routes are also ALLOTed.

Routing is <u>short-circuiting</u>.

- PSoC 5LP routing works by short-circuiting
- Shorting in-series gates: AND-gates
- Shorting in-parallel wires: OR-gates

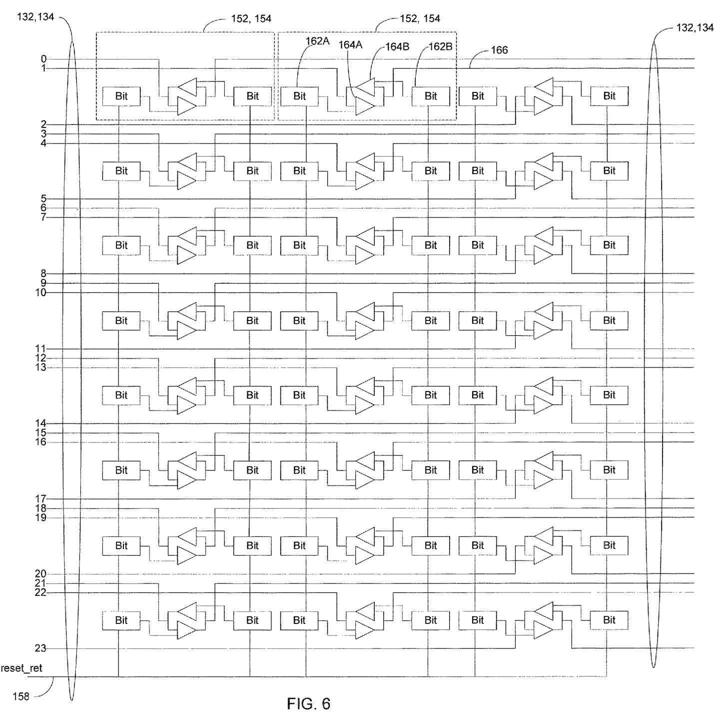


FIG. 6

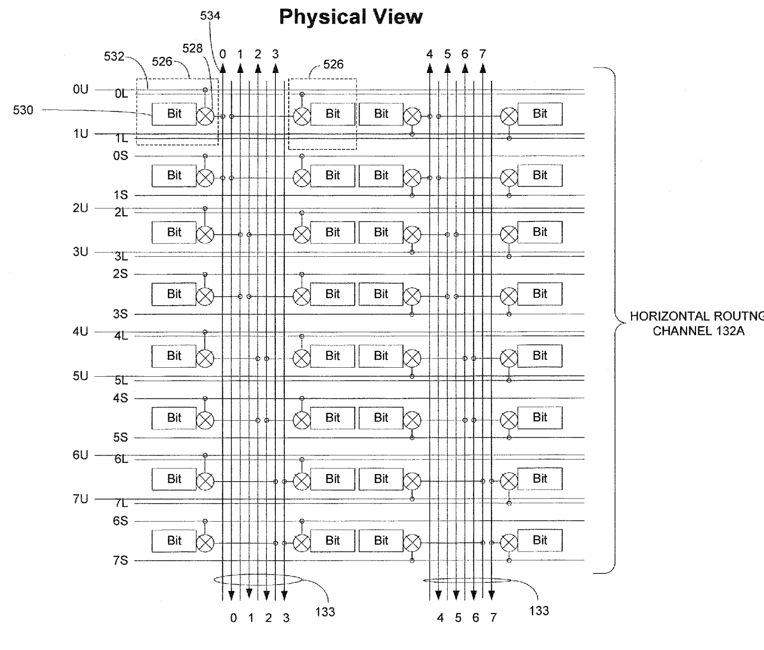
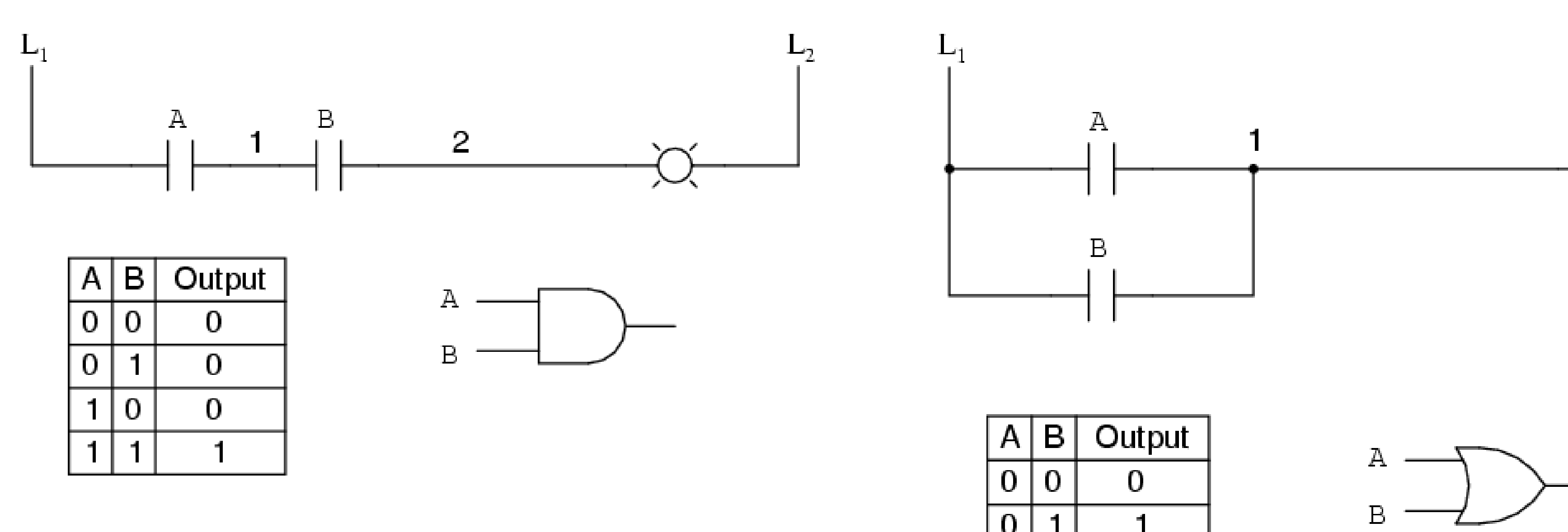


FIG. 16

Routing as logic synthesis?



A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

Placement

- How can we rearrange Boolean expressions?
- By way of the commutative, associative, distributive properties!

Done & To be done

Done:

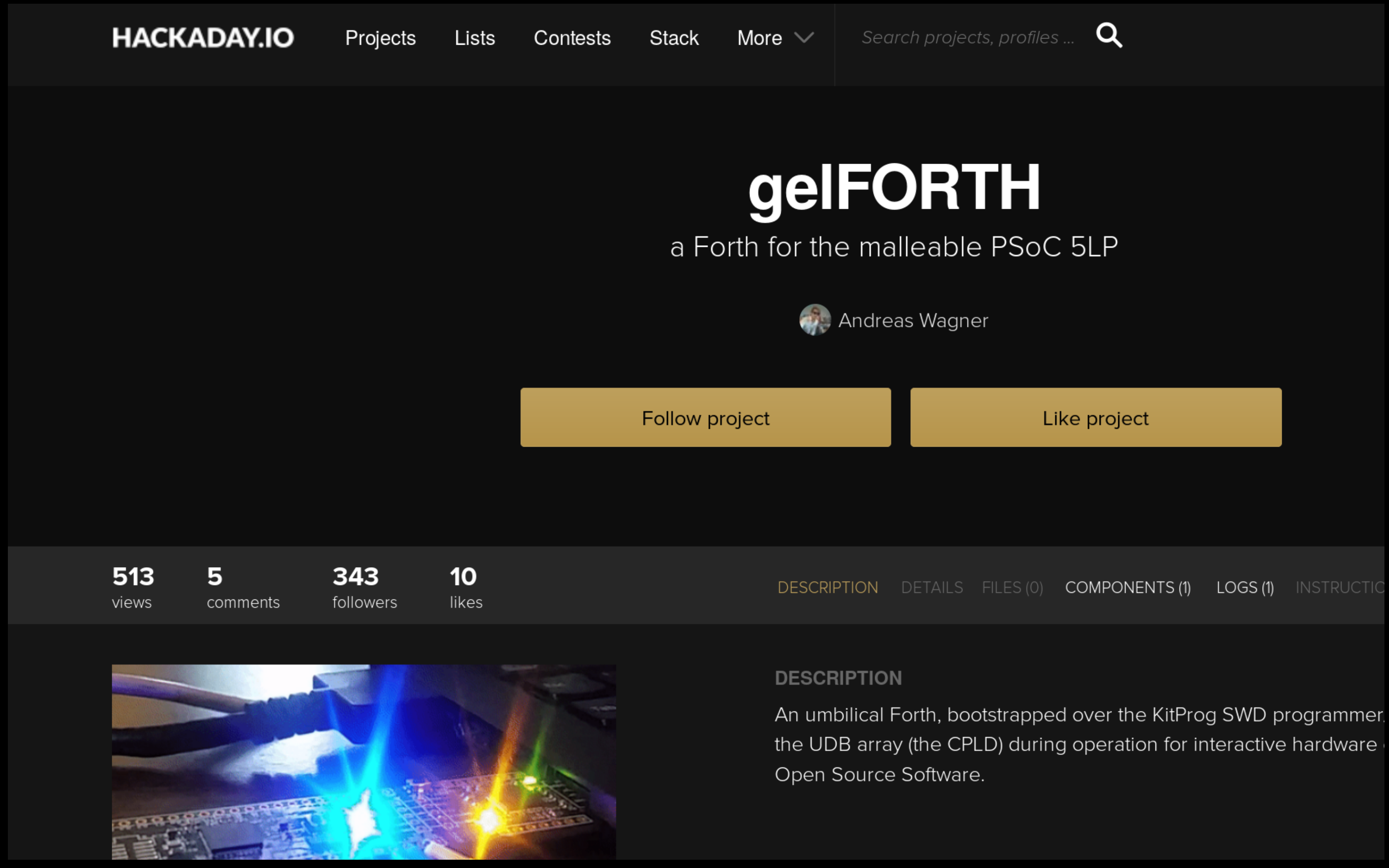
- PLD synthesis
- routing fabric PI, HS, HC, VS tiles understood

To Do:

- Parallel logic synthesis (in-progress)
- Universal Analog Blocks (in-progress)
- Hard IP blocks
- HV tiles (in-progress)

Questions , Comments , World Views?

Thanks for listening !



(– Follow the project on hackaday.io!)

references.

- https://www.reddit.com/r/Forth/comments/587dfh/high_stack_finegrain_concurrency/d8ybgi0/
- https://www.reddit.com/r/Forth/comments/599lb4/symbols_for_stack_operators/d9e7vqg/?context=9999
- http://www.forth.org/fd/FD-V21N1,2.pdf#page=21