From Idea to Hardware

High-Level Synthesis with Haskell

- Yifan Yu
- Mark Arrumm
- Rupesh Majhi

Introduction suomeksi

- Hankkeessamme olemme tutkineet erilaisia tapoja suunnitella digitaalista järjestelmää.
- Vertaamme Verilog-, C-, ja Haskell-ohjelmointikieliä, mutta Haskell-kieli on pääasiallinen tutkimuskohde.
- Esitämme Haskell-kielisen toteutuksen ja teemme havaintoesityksen.

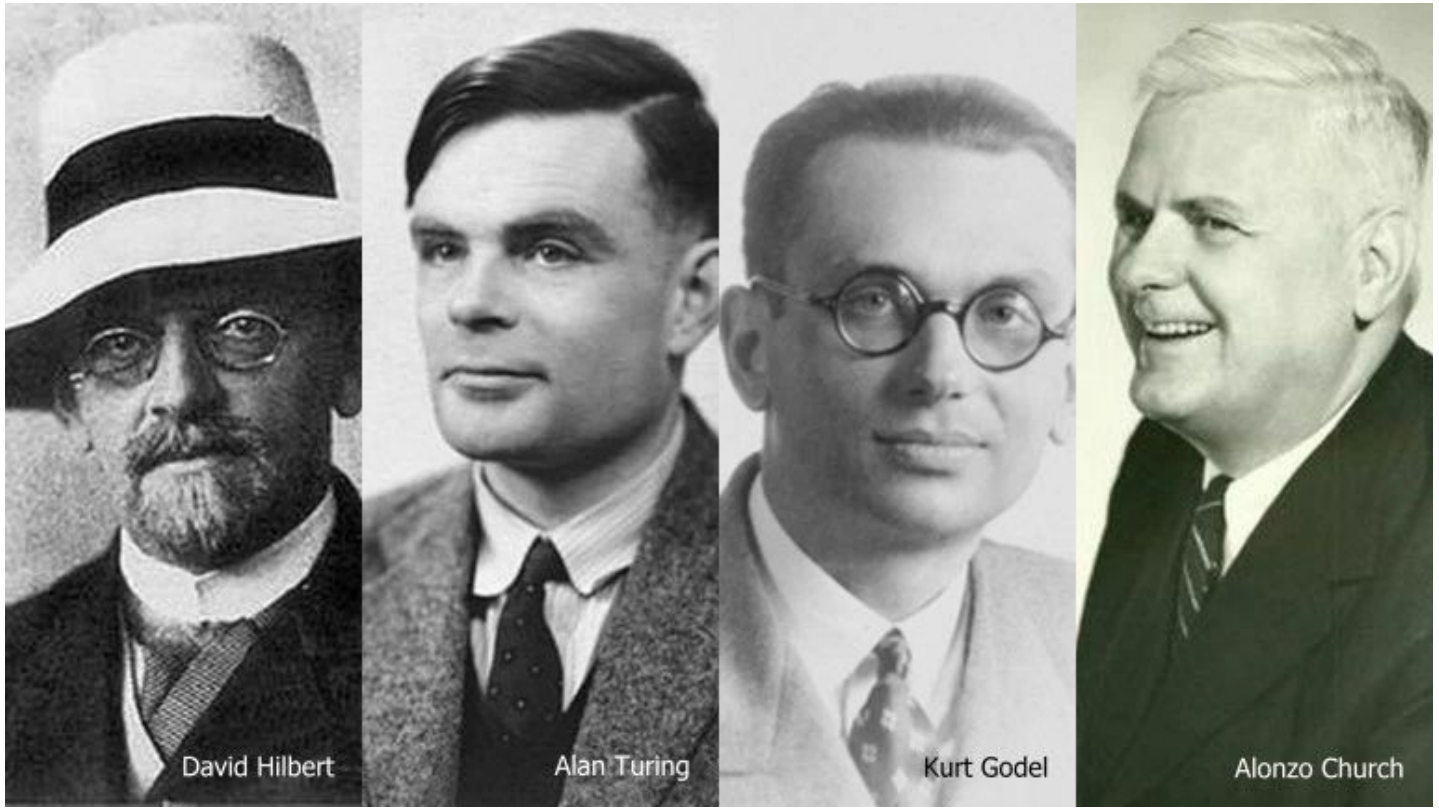
Table of Contents

Computers compute. But what is computation?

Abacus vs CPU

From a low-level point of view, an abacus does the same thing as a CPU. So hypothetically and theoretically, can we run our Windows software on top of an abacus?

Let’s talk about history!



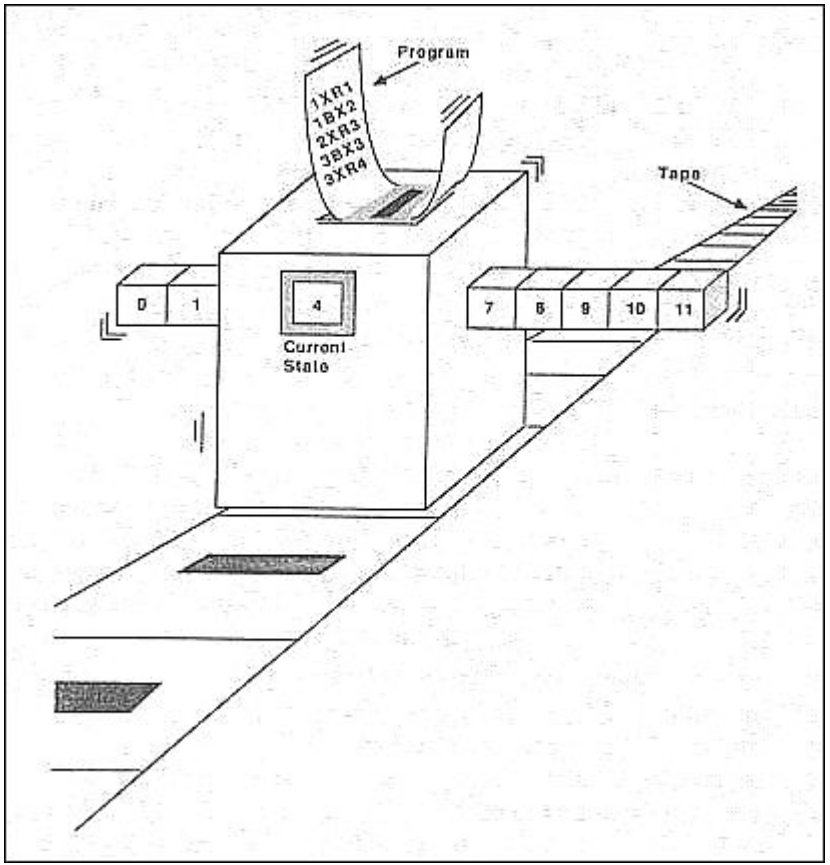
Computability

- **David Hilbert (1900)**: proof that the arithmetic is *consistent* – free of any internal contradictions (2nd problem out of 23)
- **Kurt Gödel (1931)**: it is not! (Gödel's incompleteness theorems)
  - A function is computable *if and only if* it is a *general recursive function*
- **Alan Turing (1936)**: a function is computable *if and only if* it is computable by a Turing Machine
- **Alonzo Church (1936)**: a function is computable *if and only if* it can be written down as a term of the λ-calculus

Turing machine

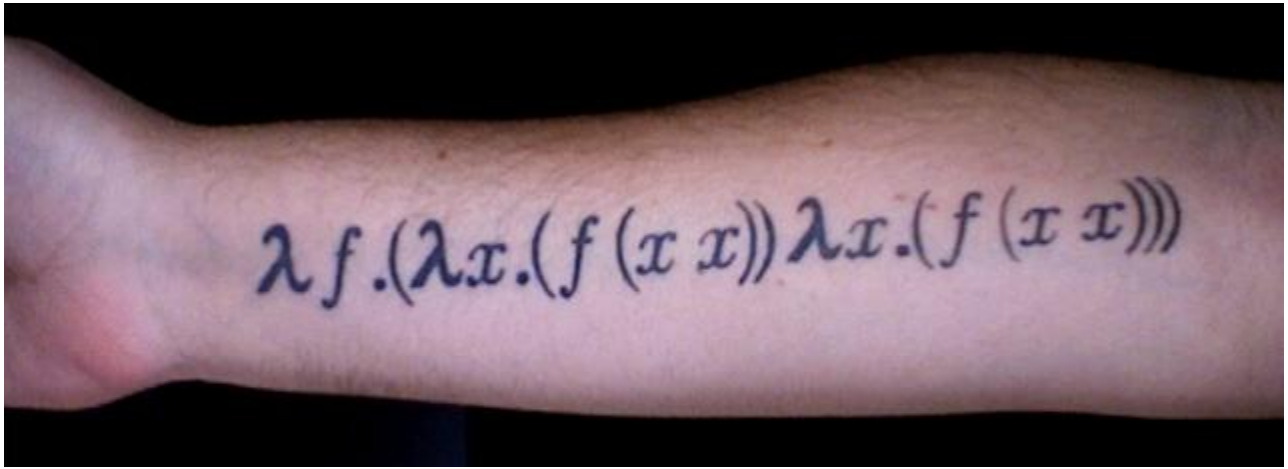
A Turing machine is an abstract machine that manipulates symbols on a strip of tape according to a table of rules.

A system is **Turing complete** if it can solve any problem that a Turing machine can.



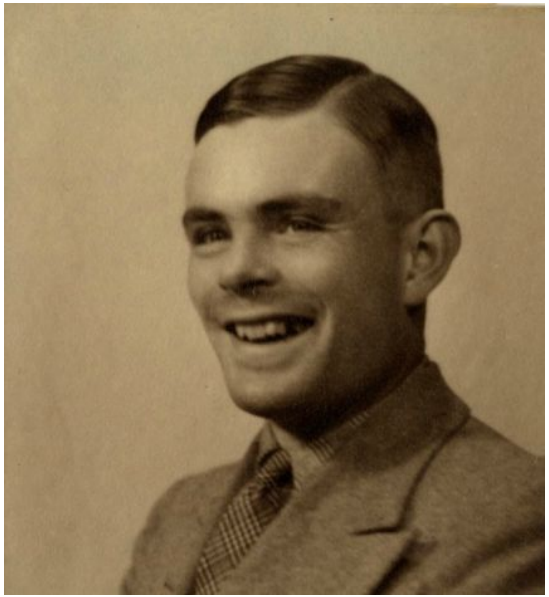
λ-calculus

- A λ-term is either:
  - A variable: x
  - Lambda abstraction: $\lambda x.\, t$
  - Application: $t\, x$



Church-Turing thesis

- They are equivalent.
- *In other words*: if there is an algorithm to solve a problem, then it can be solve with a Turing Machine or the λ-calculus
- *In other words*: a Turing Machine or the λ-calculus can simulate any mechanical computer
- *In other words*: everything feasibly computable in the physical world is feasibly computable by a Turing Machine or the λ-calculus



FP is born

- Based on λ-calculus as theoretical framework
  - Lisp in MIT, late 1950s -> lots of dialects (e.g., Scheme, Clojure)

    The final lines of NASA's LISP based launch script :

    ))))))))))))))))))))))))

    ))))))))))))))))))))))))))))))))
  - APL, early 1960s
  - ML, 1970s -> Standard ML, OCaml, F#
  - Haskell, Erlang, 1980s
  - Scala (multi-paradigm), 2004
- Mainstream languages are adding FP features: Java, C++ added lambda functions, etc.

Haskell features

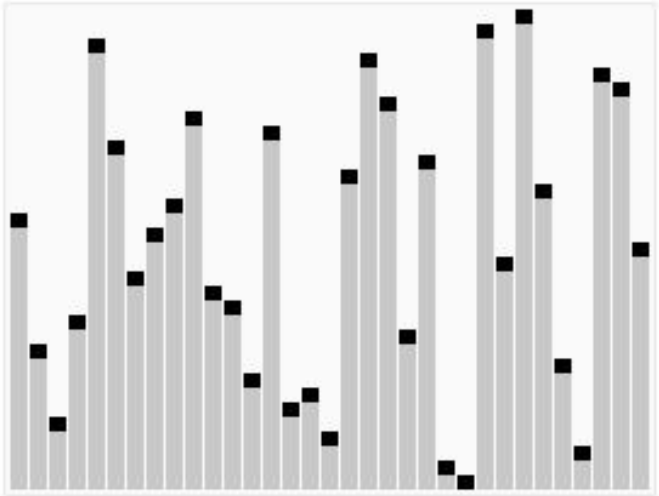
- Immutable variables
- Pure functions (no side effects)
- Higher-order functions
- Lazy evaluation
- Pattern matching
- Advanced type system
  - Full type inference
  - Strong & static typing
  - Type classes
  - Type polymorphism

Haskell Curry

A Taste of Haskell

Quicksort

```
quickSort []     = []
quickSort (x:xs) = quickSort (filter (<x) xs) ++ [x] ++ quickSort (filter (>=x) xs)
```

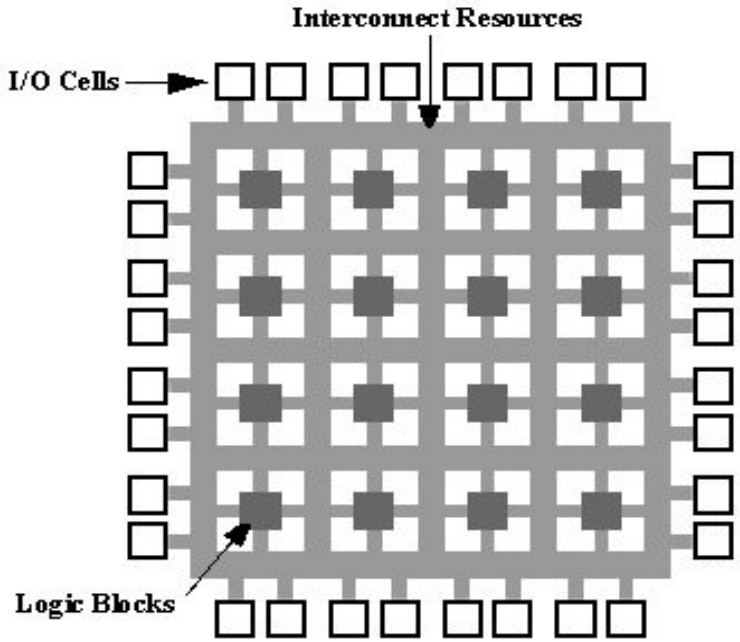


In comparison with C

```
void quicksort(int x[10],int first,int last)
{
    int pivot,j,temp,i;

     if (first < last) {
         pivot = first;
         i     = first;
         j     = last;

         while (i < j) {
             while (x[i] <= x[pivot] && i < last)
                 i++;
             while (x[j] > x[pivot])
                 j--;
```

```
            if (i < j) {
                temp = x[i];
                x[i] = x[j];
                x[j] = temp;
            }
        }

        temp     = x[pivot];
        x[pivot] = x[j];
        x[j]     = temp;
        quicksort(x, first, j-1);
        quicksort(x, j+1 , last);
    }
}
```

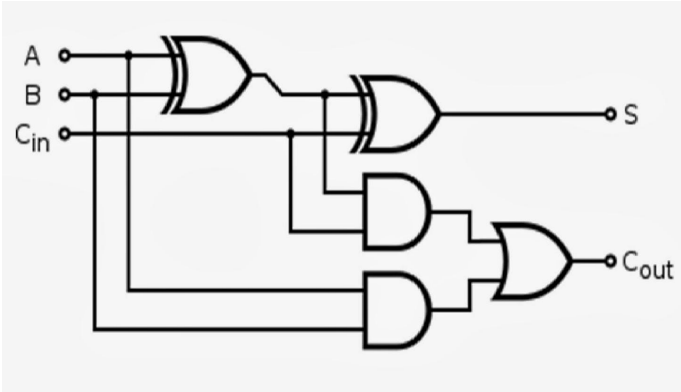
FPGA - Field Programmable Gate Array

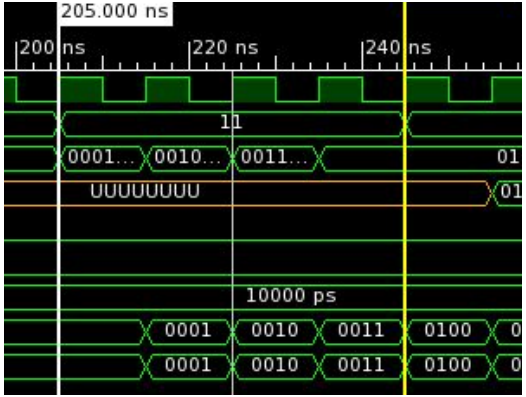


3 Ways of Describing Hardware

- What most people are using: *Verilog, VHDL*
- What some people have attempted: *HLS with C/C++/SystemC*
- What we are trying: *Cλash (Haskell based, designed by researchers in University of Twente, Netherlands)*

```
always@(posedge clk)
  if (clear == 1)
    r_out <= 0;
  else
    r_out <= r_out + a * b;
assign out = r_out;
```





Example 1: a simple OR gate

```
module or_test(
  input [1:0] sw,
  output led
  );

assign led =
  sw[0] | sw[1];

endmodule
```

Verilog

```
void or(int *a, int *b,
        int *out)
{
    *out = *a | *b;
}
```

C

```
or a b = a .|. B
```

Haskell

Example 1: a simple OR gate

```
or a b = a .|. B
```

Not only is this simpler, it also has a new feature:

```
or :: Bits a => a -> a -> a
```

Type polymorphism makes it *generic*, so that it can be applied to different types.

(not possible with Verilog or C)

Example 2: a MAC unit (multiply-accumulate)

```
module mac(
  input clk,
  input clear,
  input [15:0] a,
  input [15:0] b,
  output [15:0] out)
  reg [15:0] state;

  always@(posedge clk)
    if (clear == 1)
      state <= 0;
    else
      state <= state + a * b;
  assign out = state;
endmodule
```

Verilog

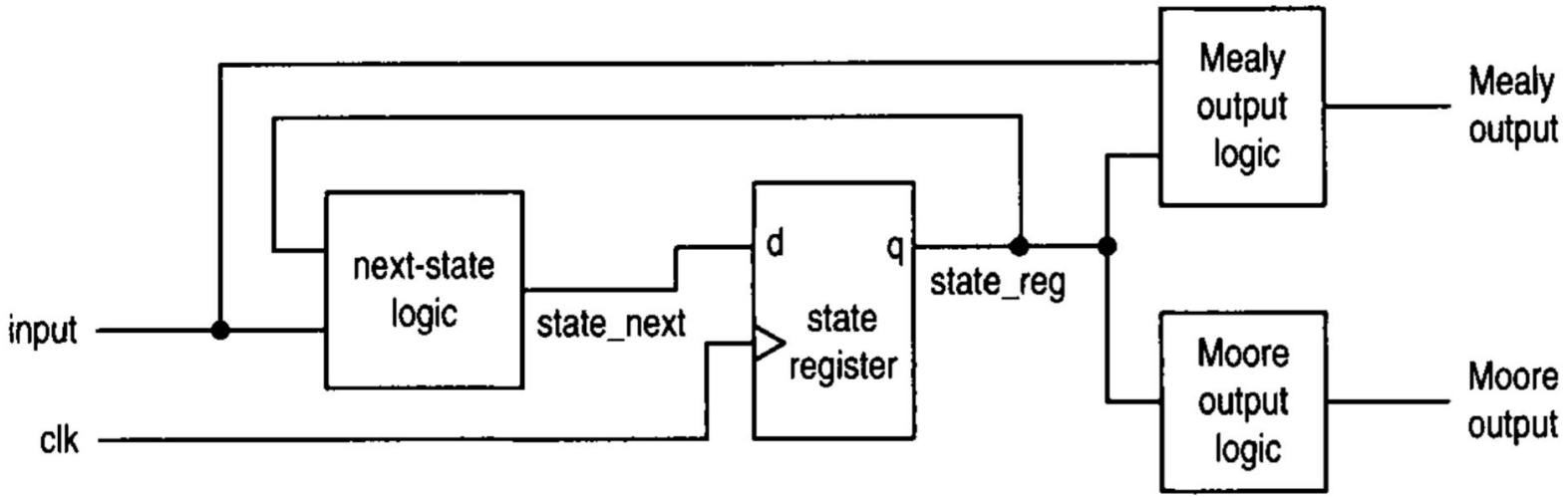
```
void macc(int a, int b,
          int *accum, bool clr)
{
   static acc_reg = 0;
   if (clr)
      acc_reg = 0;
   acc_reg += a * b;
   *accum = acc_reg;
}
```

C

```
macT acc input = (f acc input, acc)
  where
  f a (x, y) = a + x * y

mac = mealy macT (0, 0)
```

Haskell

Mealy machine vs Moore machine



The higher-order `mealy` function

```
mealy :: (s -> i -> (s, o))      state -> in -> (newstate, out)
      -> s                       initial state
      -> (Signal i -> Signal o)  synchronous sequential function
```

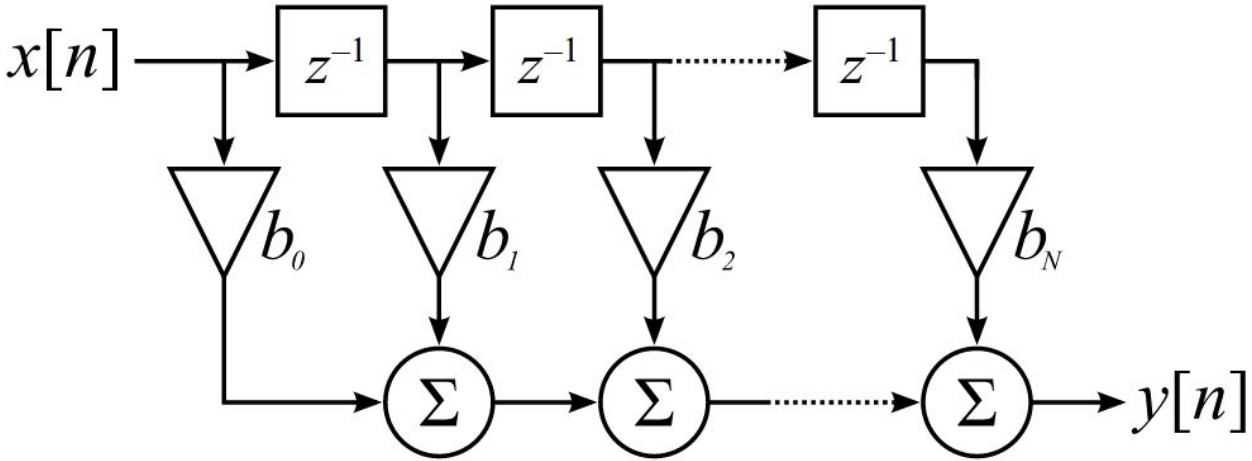
The MAC unit we saw earlier:

```
macT acc input = (f acc input, acc)
  where
  f a (x, y) = a + x * y

mac = mealy macT (0, 0)
```

Higher-order function **abstracts away** some of the common patterns, so you can construct any Mealy machine with a pure state transition function.

Example 3: an FIR filter (finite impulse response)

- Verilog & C implementation are too long to show here
- FIR filter definition: the dot-product of a set of filter coefficients and a window over the input



Example 3: an FIR filter (finite impulse response)

- Verilog & C implementation are too long to show here
- FIR filter definition: the dot-product of a set of filter coefficients and a window over the input

Haskell

```
fir coeffs input = dotp coeffs (window input)
  where
    dotp as bs = sum (zipWith (*) as bs)
```

Notice how closely the code matches the definition!

A hierarchy of abstractions

Software design: machine code -> assembly code -> C -> OOP, FP, etc

Hardware design: HDL -> HLS (C -> FP)

```
01110011 01
01100101 01
01101000 01
01100100 01
01110010 01
01110100 01
01100001 01
01101001 01
01101101 01
00100000 01
01110011 01
```

```
push    ebp
mov     ebp, esp
movzx   ecx, [ebp
pop     ebp
movzx   dx, cl
lea     eax, [edx
add     eax, edx
shl     eax, 2
add     eax, edx
shr     eax, 8
sub     cl, al
shr     cl, 1
add     al, cl
```

```
--------------------Monte Carl
while(t1<=tmax)
  {
      fprintf(fp,"%f\t%d\n",t1,
      t1+=dt;
      for(j=0; j<=n0; j++)
     {
              r=rand()%1000;
              r=r/1000;
              if(r<=0.001) n0--;
              if(n0<0) goto l1;
     }
  }
--------------------------------
fclose(fp);
```

```
-- Type annotation (optional)
fib :: Int -> Integer

-- With self-referencing data
fib n = fibs !! n
        where fibs = 0 : scanl (
        -- 0,1,1,2,3,5,...

-- Same, coded directly
fib n = fibs !! n
        where fibs = 0 : 1 : nex
              next (a : t@(b:_))
```

Circuit vs Haskell

Circuit

- Combinational circuits
- Undefined behavior OK if you don't use it
- D flip-flop "delays" a signal by one cycle
- Self-reference

Haskell

- Pure functions
- Undefined value ok if you don't use it (lazy evaluation)
- (:) "delays" everything in a list by one element
- Self-reference

Now, let’s design something in Haskell!

- It has to be sufficiently sophisticated
- At the same time, simple enough for us to pick up CλaSH and finish the design in just few weeks
- It has to reflect issues we need to address in real-world projects
- It has to be fun

A Brainfuck CPU

What the Brainfuck?

- A minimalism programming language with an ungodly name
- Has only 8 commands: > < + - . , [ ]
- Everything else gets ignored
- Operates on an array of bytes
- **It is fully Turing-complete**
- In face, it is very close to a Turing Machine

Show me the code!

```
++++++++++[>+++++++>++++++++++>+++>+
<<<<-]>++.>+.+++++++..+++.>++.<<+++
++++++++++.>.+++.------.--------.>+
.>.
```

That's the "Hello, World!" Program

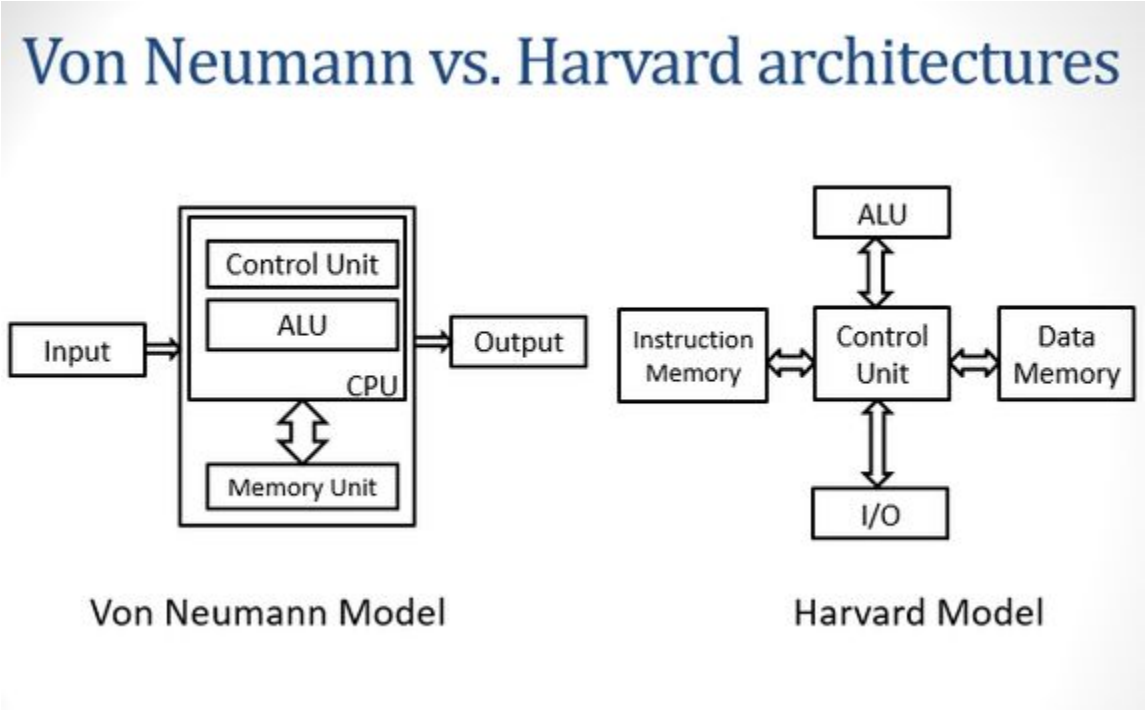
So... it’s pretty useless, why make a CPU for it?

- Software interpreter is trivial to implement, CPU is harder
- Simple instructions, but demonstrates the main principles of CPU design
- We need to address common issues
  - Architecture choice
  - Instruction timing
  - Caches
- Perfect for testing:
  - the capabilities of CλaSH
  - the feasibility of describing hardware with Haskell
- It sounds like fun!

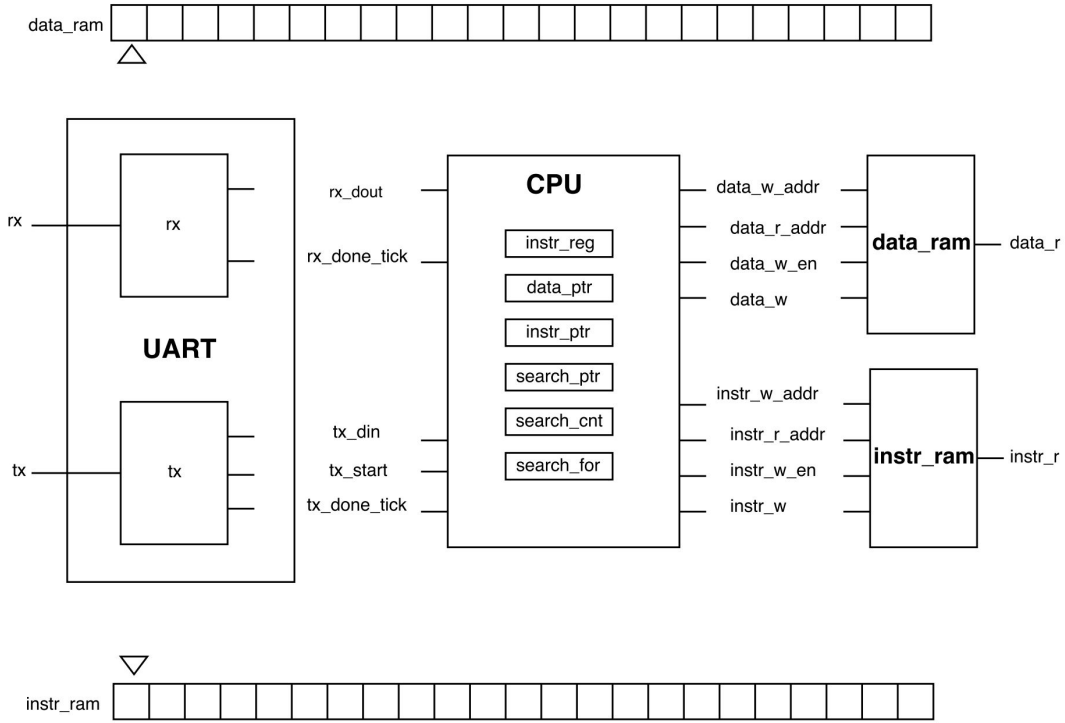
What's new in our design?

- The idea of a Brainfuck CPU is not new. Different approaches:
  - VHDL
  - Lava (Haskell-based DSL)
  - Python + MyHDL
  - Directly with logic gates
- Our CλaSH-based design compiles directly from Haskell to Verilog
  - Enables testing and debugging with REPL
  - High-level constructs are used such as `mealy`
- Directly programmable through UART
- Counts clock cycles and displays on seven-segment display
- Has a debugging mode which enables clock-by-clock debugging
- A number of optimizations are exploited

Design: architecture



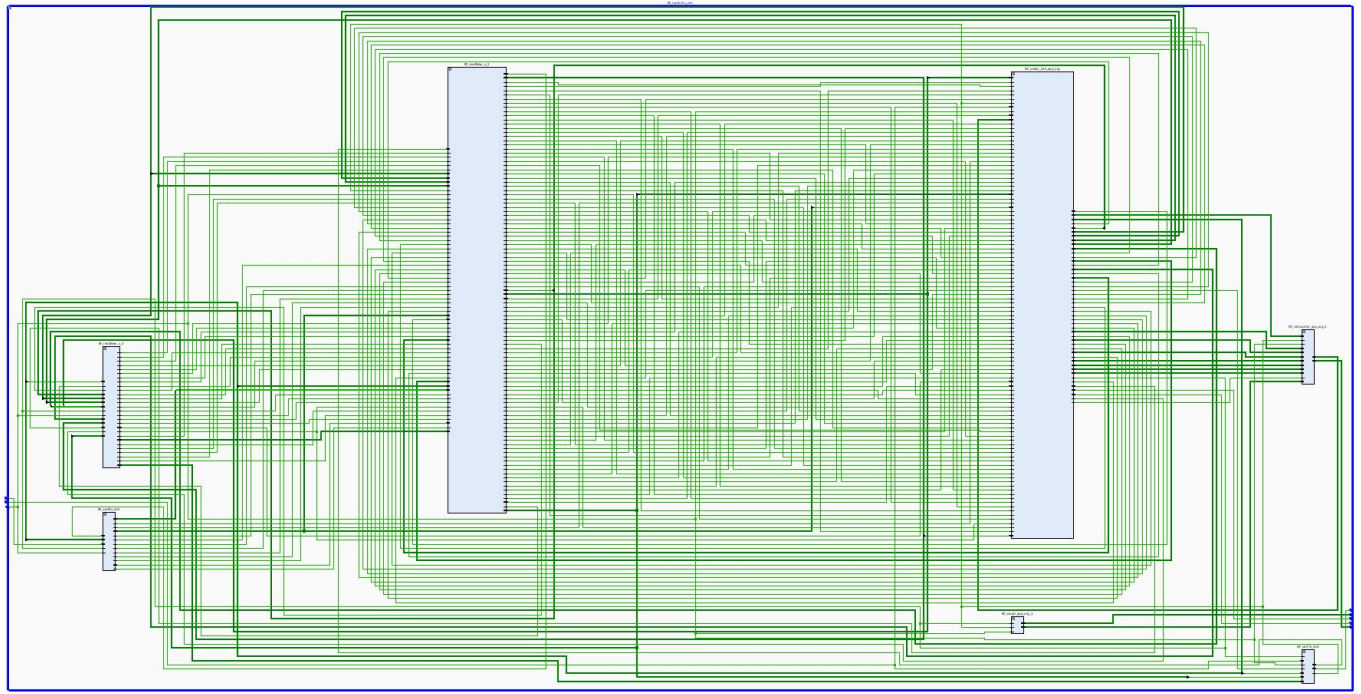
Design: Version 1



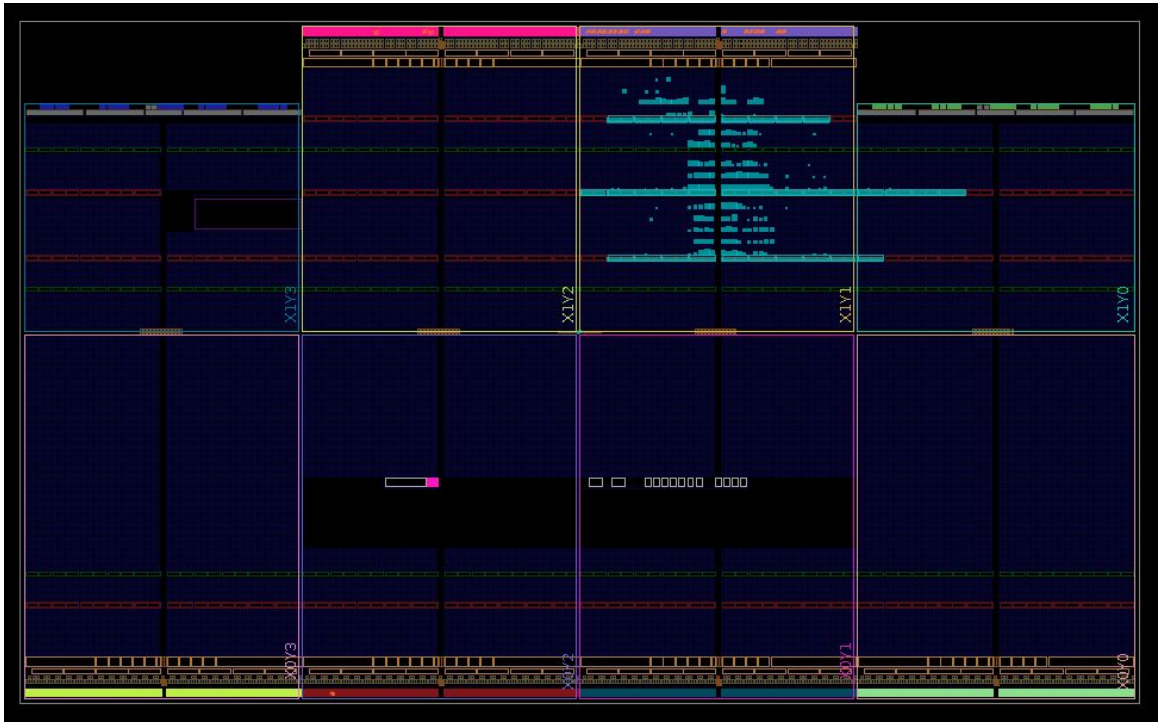
Results

- Version 1: **648** lines of Haskell (including SSeg, UART) generated **3036** lines of Verilog
- Version 1 is able to run test programs, *nicely and slowly*
- More optimizations can be exploited
  - Jump address pre-indexing
  - Common instruction patterns can be reduced to a single instruction
  - Pipelining
- More updates on GitHub: https://github.com/aufheben/clash-bfcpu

Synthesized Schematic



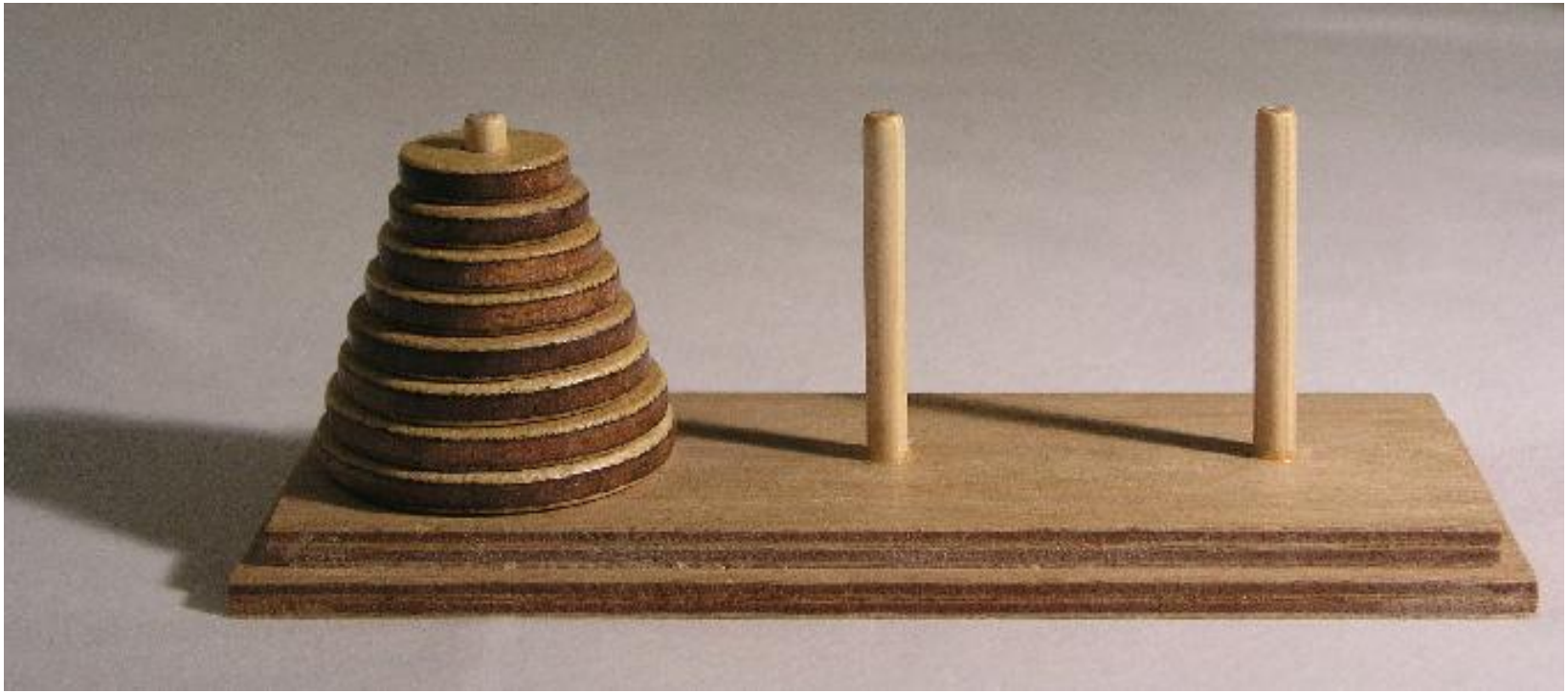
Implementation



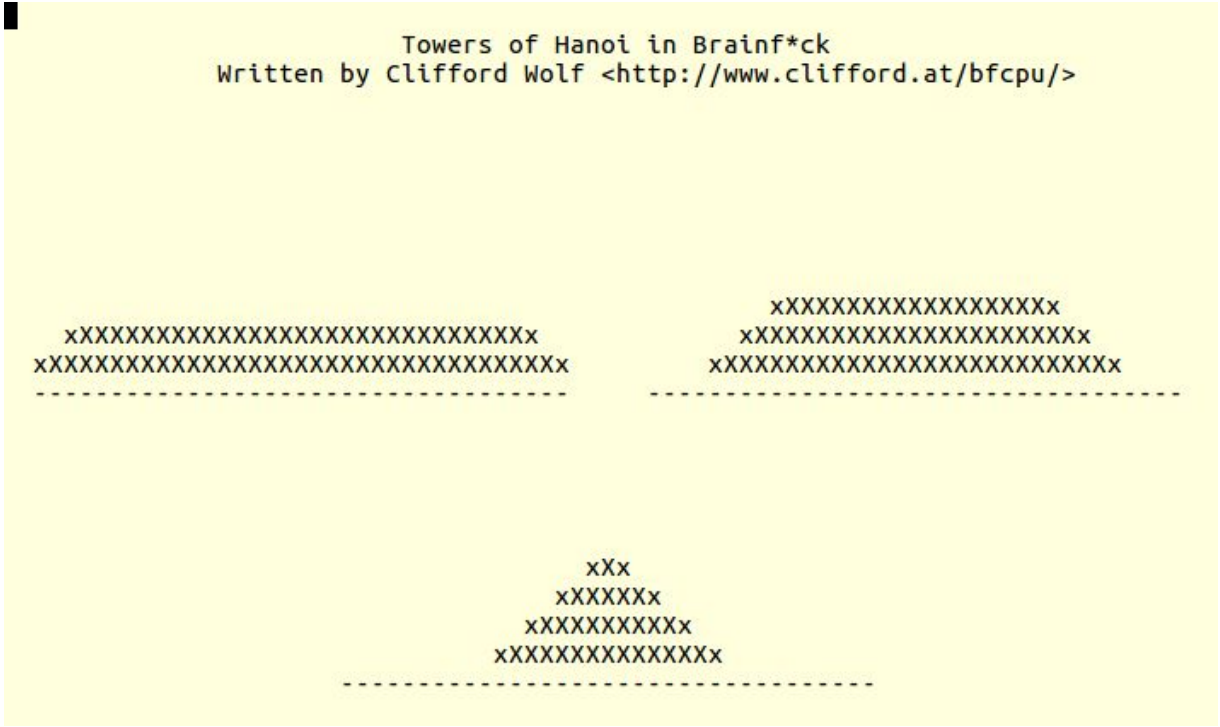
Now, live demonstration!

```
++++++++++[>+++++++>++++++++++>+++>+
<<<<-]>++.>+.+++++++..+++.>++.<<++++
++++++++++.>.+++.------.--------.>+
.>.
```

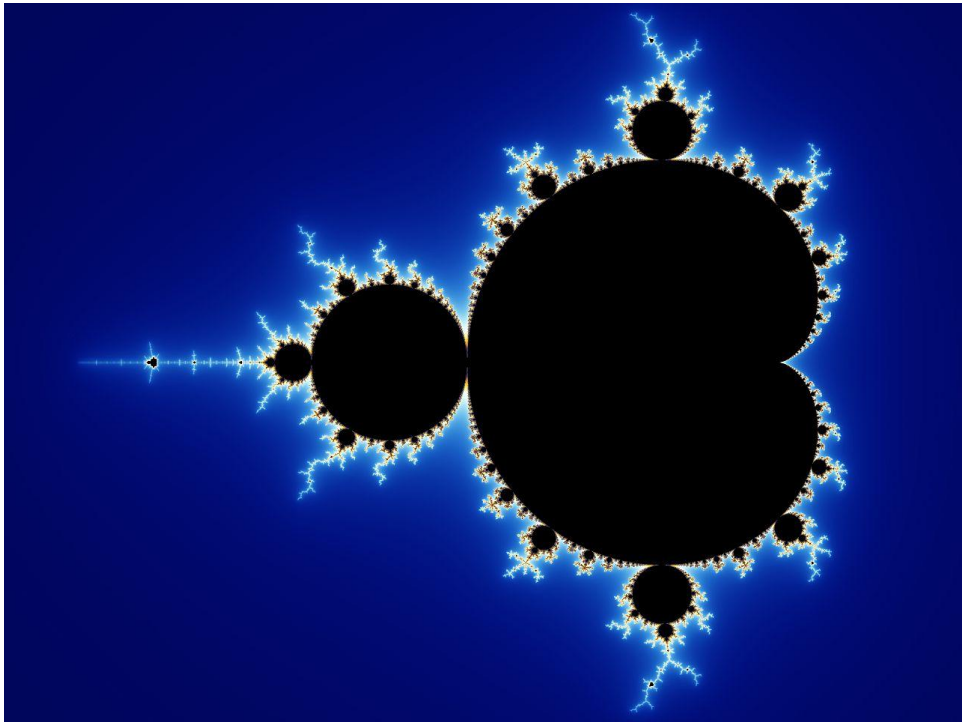
Tower of Hanoi *(click to download)*



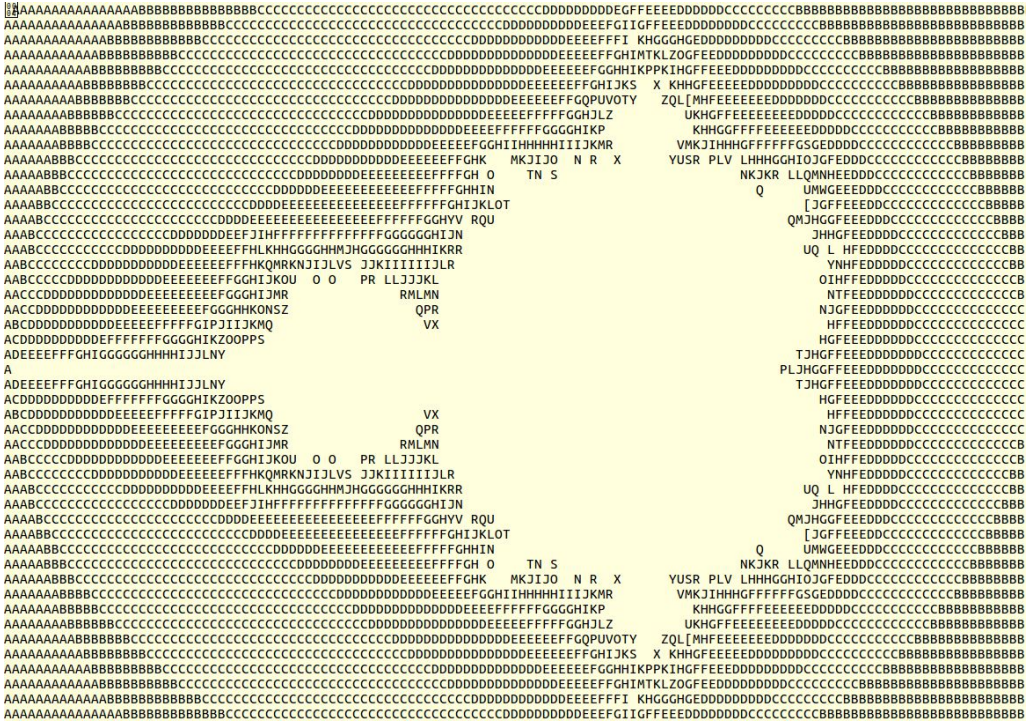
[Screenshot From Live Demo]



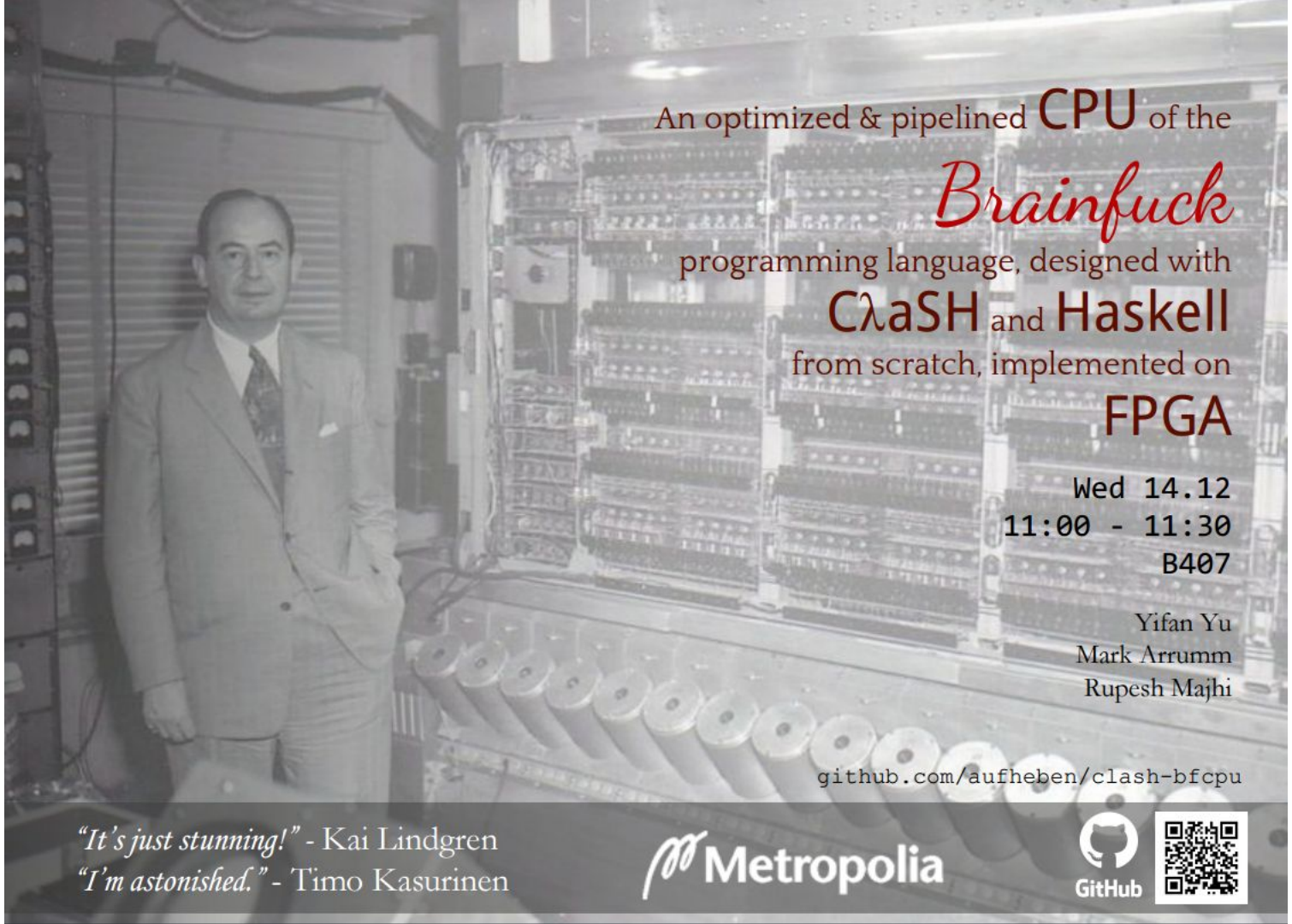
Mandelbrot set *(click to download)*



[Screenshot From Live Demo]



Try: Brainfuck Interpreter Written in Brainfuck


An optimized & pipelined CPU of the
Brainfuck
programming language, designed with
CλaSH and Haskell
from scratch, implemented on
FPGA
Wed 14.12
11:00 - 11:30
B407
Yifan Yu
Mark Arrumm
Rupesh Majhi
github.com/aufheben/clash-bfcpu
"It's just stunning!" - Kai Lindgren
"I'm astonished." - Timo Kasurinen
Metropolia
GitHub