Designing Circuits with Code

Nathan SOUFFLET

A very manual design process

2.2 Architecture

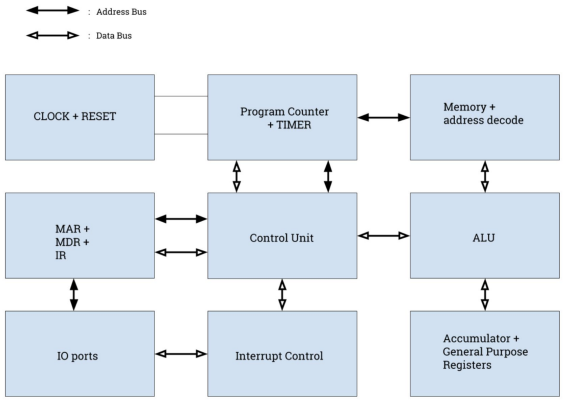


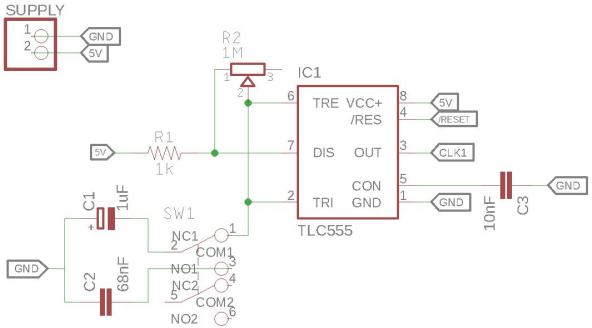
FIGURE 2 – Organisation possible des modules

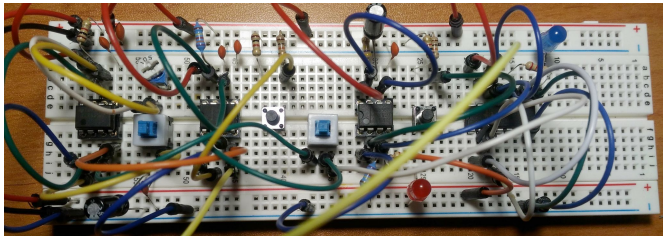
2.2.1 CLOCK + RESET

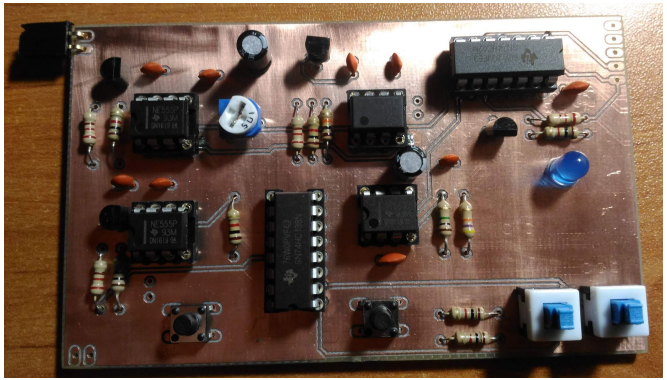
Ce module s'occupe de la génération d'un signal carré périodique permettant de synchroniser les opérations des différents modules entre-eux. Il dispose de trois modes d'opération :

- Lent : Permet de faire varier la fréquence entre 0.8 et 400Hz
- Rapide : Permet de faire varier la fréquence entre 1kHz et 1.6MHz
- Manuel : Permet de contrôler manuellement l'horloge (mode pas à pas)

La seconde fonction de ce module est de réinitialiser le processeur pendant la première demi-seconde après allumage (de sorte à ce que la tension se stabilise à 5V garantissant le bon fonctionnement des modules) et à chaque appui d'un bouton dédié.







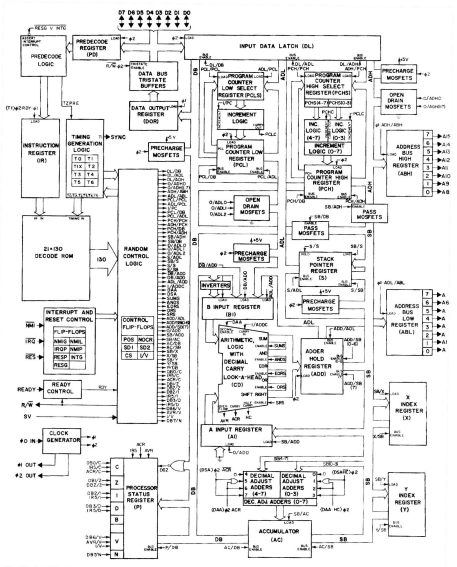
Specification (What/Why)

Logic Design (How)

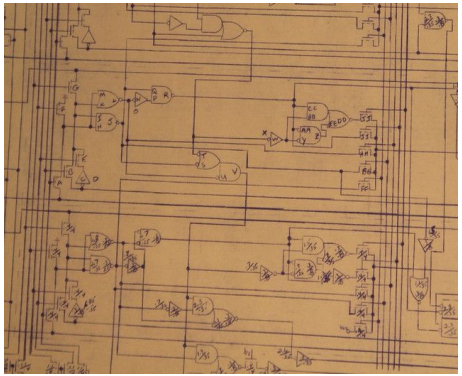
Physical Design (Where)

The Chip Design Process in the 70s

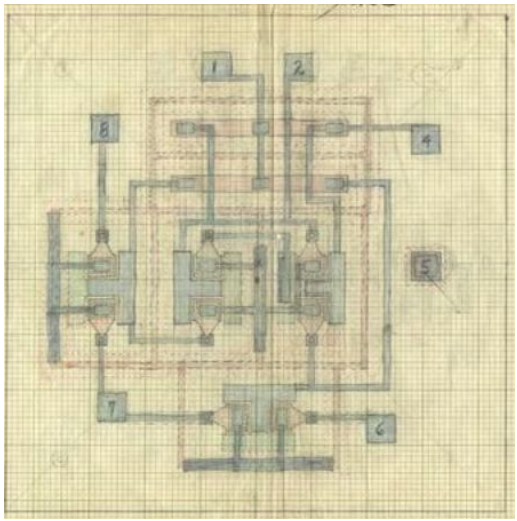
Module level



Gate level



Physical layout



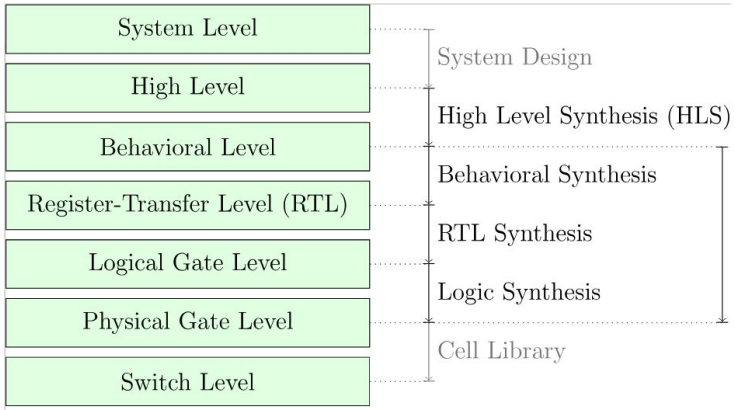
Challenges

- Iterations are very slow
- Manual errors can be introduced at each step
- Design verification/testing is difficult

Rubylith operators (Intel)

Electronic Design Automation (EDA)

Goal: Reason about a circuit design at a **high level of abstraction** and let tools handle the low-level transformations (logic synthesis, physical routing, equivalence verification, …)



Computervision CADDS3 system

Electronic Design Automation (EDA)

Goal: Reason about a circuit design at a **high level of abstraction** and let tools handle the low-level transformations (logic synthesis, physical routing, equivalence verification, …)

Benefits of high-level representations:

- **Easier to reason** about circuits, both for humans and computer -> More optimisation potential

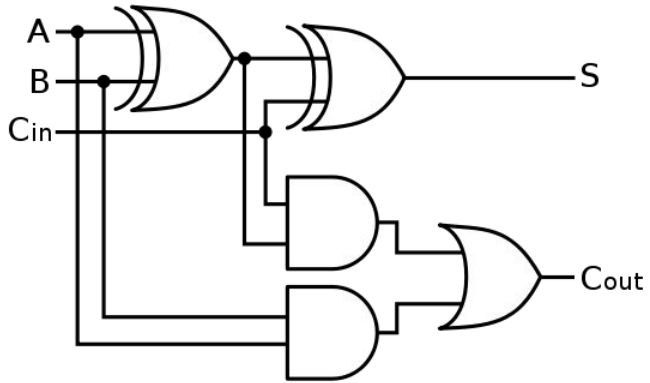
- More **efficient simulation** (Direct mapping to high-level programming languages)

Computervision CADDS3 system

Describing Hardware

```
module FullAdder(
    a: bool,
    b: bool,
    carry_in: bool,
) -> (
    sum: bool,
    carry_out: bool,
) {
    let a_xor_b = a xor b
    sum = carry_in xor a_xor_b
    carry_out = (carry_in and a_xor_b) or (a and b)
}
```

Yet anOther hardware Description Language (YODL)



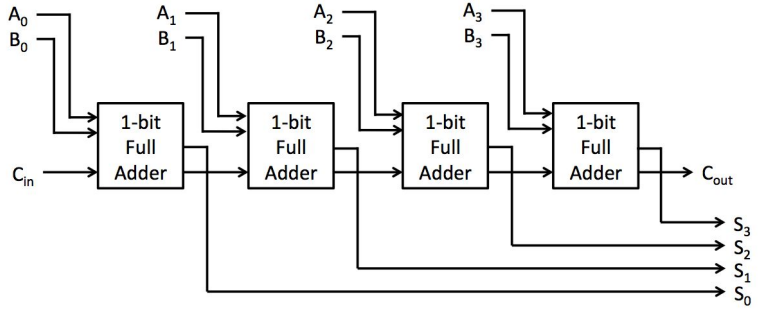
Visual representation

Parametric modules: Circuit Generators

```
module Adder<N: uint>(
    a: uint<N>,
    b: uint<N>,
    carry_in: bool,
) -> (
    sum: uint<N>,
    carry_out: bool,
) {
    let carry_chain: bool[N + 1]
    carry_chain[0] = carry_in
    let bits: bool[N]

    for i in 0..<N {
        FullAdder(
            a: a[i],
            b: b[i],
            carry_in: carry_chain[i],
            carry_out: carry_chain[i + 1],
            sum: bits[i],
        )
    }

    carry_out = carry_chain[N]
    sum = uint!(bits)
}
```



Code as a flexible representation

Hardware synthesis becomes a standard compilation process

```
module Adder<N: uint>(
    a: uint<N>,
    b: uint<N>,
    carry_in: bool,
) -> (
    sum: uint<N>,
    carry_out: bool,
) {
    let carry_chain: bool[N + 1]
    carry_chain[0] = carry_in
    let bits: bool[N]

    for i in 0..<N {
        FullAdder(
            a: a[i],
            b: b[i],
            carry_in: carry_chain[i],
            carry_out: carry_chain[i + 1],
            sum: bits[i],
        )
    }

    carry_out = carry_chain[N]
    sum = uint!(bits)
}
```

Remove metaprogramming constructs

```
module Adder$0(
  a: uint<32>,
  b: uint<32>,
  carry_in: uint<1>,
) -> (
  sum: uint<32>,
  carry_out: uint<1>,
) {
  let carry_chain: uint<1>[33];
  carry_chain[0] = carry_in;
  let bits: uint<1>[32];
  FullAdder(
    a: a[0],
    b: b[0],
    carry_in: carry_chain[0],
    carry_out: carry_chain[1],
    sum: bits[0],
  );
  FullAdder(
    a: a[1],
    b: b[1],
    carry_in: carry_chain[1],
    carry_out: carry_chain[2],
    sum: bits[1],
  );
  FullAdder(
    a: a[2],
```

Remove high-level constructs

```
wire \xor1
wire $xor_0.y
wire $or_1.y
wire $and_2.y
wire $and_3.y
wire $xor_4.y
cell $xor $xor_0
  parameter \A_SIGNED 0
  parameter \B_SIGNED 0
  parameter \A_WIDTH 1
  parameter \B_WIDTH 1
  parameter \Y_WIDTH 1
  connect \A \carry_in
  connect \B \xor1
  connect \Y $xor_0.y
end
cell $and $and_2
  parameter \A_SIGNED 0
  parameter \B_SIGNED 0
  parameter \A_WIDTH 1
  parameter \B_WIDTH 1
  parameter \Y_WIDTH 1
  connect \A \carry_in
  connect \B \xor1
```

Behavioural Description

Multibit Gate Level

Abstraction Level Lowering = Compiler Pass

```
module Top(clk: clock, rst: bool) -> (q: uint<8>) {
    let reg = Reg<{ x: uint<8>, y: uint<8> }>(clk, rst)
    reg.d.x = reg.q.x + 1

    if (reg.q.x == 8'hFF) {
        reg.d.y = reg.q.y + 1
    }

    q = reg.q.x + reg.q.y
}
```

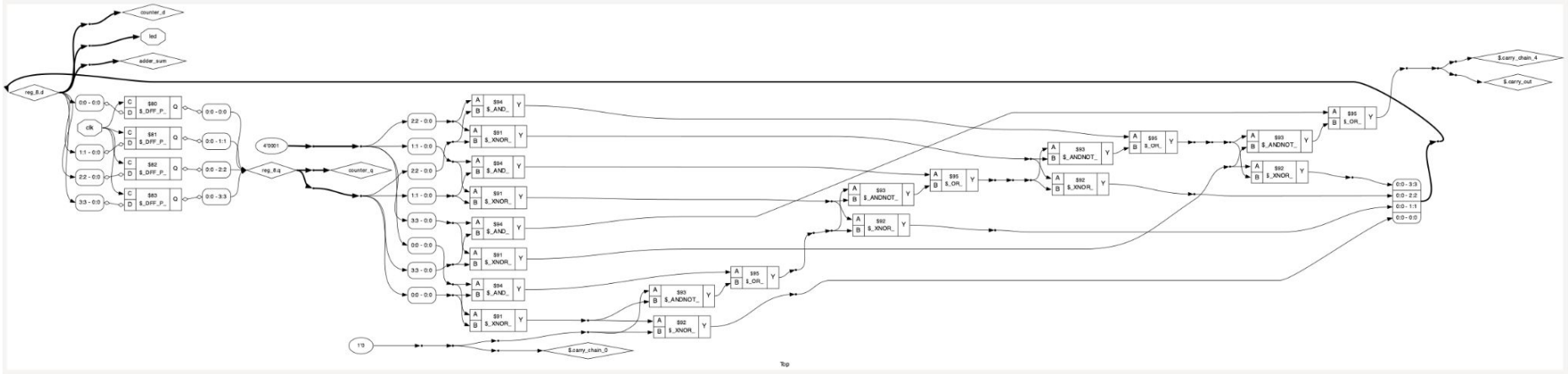


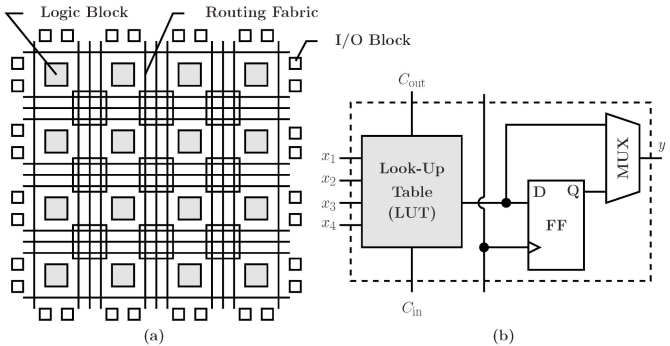
```
module Top(clk: clock, rst: bool) -> (q: uint<8>) {
    let reg_x = Reg<uint<8>>(clk, rst)
    let reg_y = Reg<uint<8>>(clk, rst)
    reg_x.d = reg_x.q + 1
    reg_y.d = reg_x.q == 8'hFF ? reg_y.q + 1 : reg_y.q
    q = reg_x.q + reg_y.q
}
```

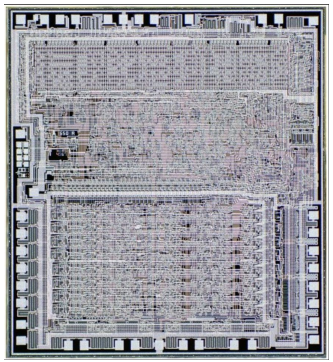
Main passes:

- **Monomorphisation** (Remove parametric modules)
- **Type Checking**
- **Optimisations** (Constant Folding, Common Subexpression Elimination, …)
- **Lower aggregate types** (structs, vectors, …) to ground types (uint, sint)
- Lower **conditional assignments to multiplexer** trees
- **Lower high-level operations** (add, mul, mux) to primitive logic gates
- **Flatten** module hierarchies (inlining)

Technology Mapping







FPGA

ASIC

Yodl Playground

Example: RISCV.yodl Pass: write_low_firrtl Compile System

Input

```
54
55  module CPU(
56      clk: clock,
57      rst: bool,
58      mem_read_data: uint<32>,
59      mem_read_busy: bool,
60  ) -> (
61      mem_addr: uint<32>,
62      mem_read_enable: bool,
63      mem_write_data: uint<8>[4],
64      mem_write_mask: bool[4],
65      status: uint<24>,
66  ) {
67      let inst = Reg<uint<32>>(clk, rst)
68      let opcode = inst.q[6:0]
69      let is_alu_reg = opcode == Inst::ALU_REG
70      let is_alu_imm = opcode == Inst::ALU_IMM
71      let is_branch = opcode == Inst::BRANCH
72      let is_jalr = opcode == Inst::JALR
73      let is_jal = opcode == Inst::JAL
74      let is_auipc = opcode == Inst::AUIPC
75      let is_lui = opcode == Inst::LUI
76      let is_load = opcode == Inst::LOAD
77      let is_store = opcode == Inst::STORE
78      let is_system = opcode == Inst::SYSTEM
79
80      let imm_u = {inst.q[31:12], 12'0}
81      let imm_i = {21*{inst.q[31]}, inst.q[30:20]}
82      let imm_s = {21*{inst.q[31]}, inst.q[30:25], inst.q[11:7]}
83      let imm_b = {20*{inst.q[31]}, inst.q[7], inst.q[30:25], inst.q[11:8], 1'b0}
84      let imm_j = {12*{inst.q[31]}, inst.q[19:12], inst.q[20], inst.q[30:21], 1'b0}
85
86      let stage = Reg<uint<2>>(clk, rst)
87      let fresh_inst = stage.q == Stage::WAIT_INST ? mem_read_data : inst.q
88      let rs1_addr = fresh_inst[19:15]
89      let rs2_addr = fresh_inst[24:20]
90      let rd_addr = inst.q[11:7]
91
92      let funct3 = inst.q[14:12]
93      let funct7 = inst.q[31:25]
94
95      let pc = Reg<uint<32>>(clk, rst)
96      let rs1 = Reg<uint<32>>(clk, rst)
97      let rs2 = Reg<uint<32>>(clk, rst)
98      let regs = RegisterFile(clk, rs1_addr, rs2_addr, rd_addr)
```

Output

```
1   FIRRTL version 4.1.0
2   circuit CPU:
3     public module RegisterFile:
4       input clk: Clock
5       input rs1_addr: UInt<5>
6       input rs2_addr: UInt<5>
7       input rd_addr: UInt<5>
8       input rd_data: UInt<32>
9       input write_enable: UInt<1>
10      output rs1: UInt<32>
11      output rs2: UInt<32>
12
13      wire regs_q_0: UInt<32>
14      wire regs_q_1: UInt<32>
15      wire vec_0$_0_clk: Clock
16      wire vec_0$_0_en: UInt<1>
17      wire vec_0$_0_addr: UInt<5>
18      wire vec_0$_1_clk: Clock
19      wire vec_0$_1_en: UInt<1>
20      wire ve
21      wire st
22      wire st
23      wire st
24      wire st
25      wire st
26      wire st
27      wire ve
28      wire ve
29      wire ve
30      wire ve
31      wire ve
32      wire st
33      wire st
34      wire st
35      wire st
36      wire st
37      mem mem
38        data-
39        depth
40        read-
41        write
42        read-
43        reade
44        reade
45        write
```

CPU 16

Simple load-store 16-bit CPU ISA, along with an assembler, compiler, simulator and Yodl design.