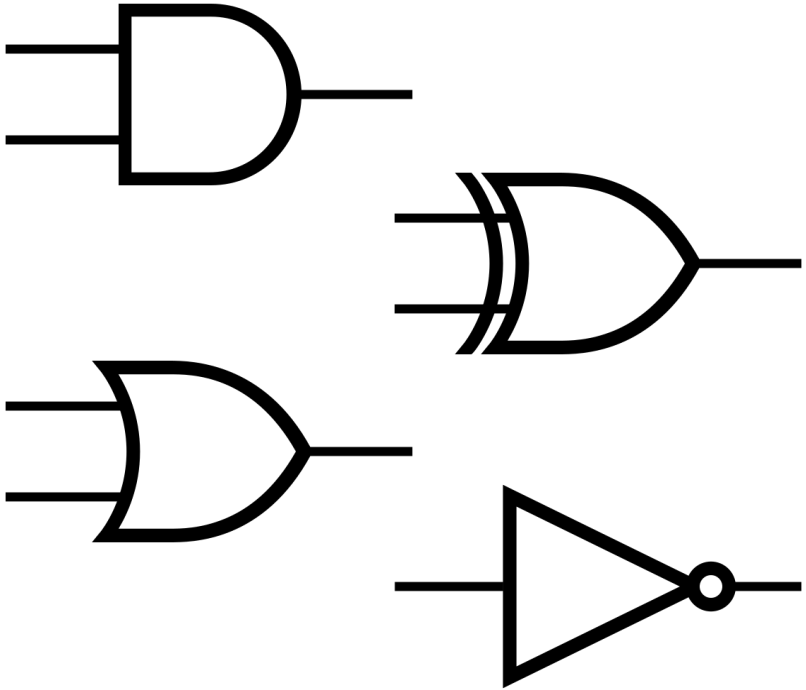
FPGA CPU Design

Graham Edgecombe

Combinational logic

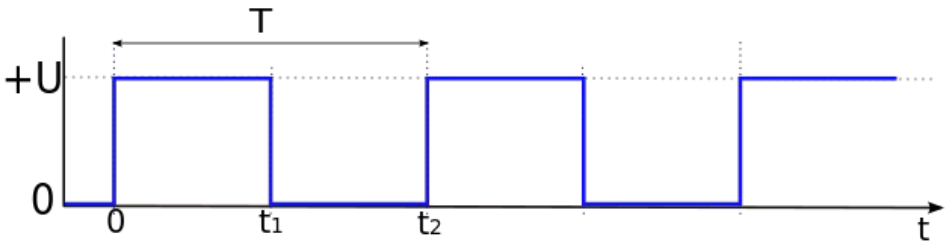
- Output only depends on current inputs
- No loops, memory or state
- Example: AND gate

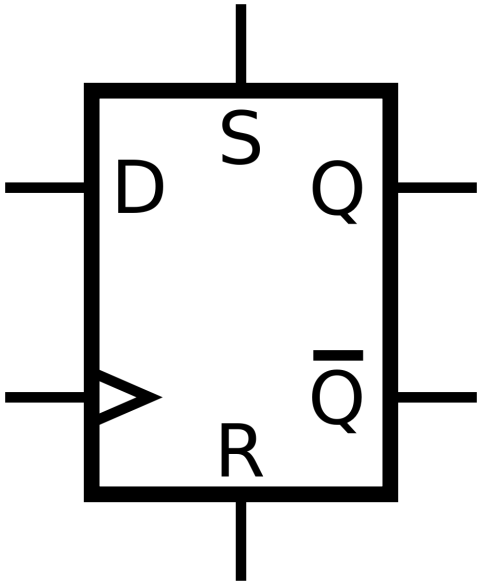
A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1



Synchronous sequential logic

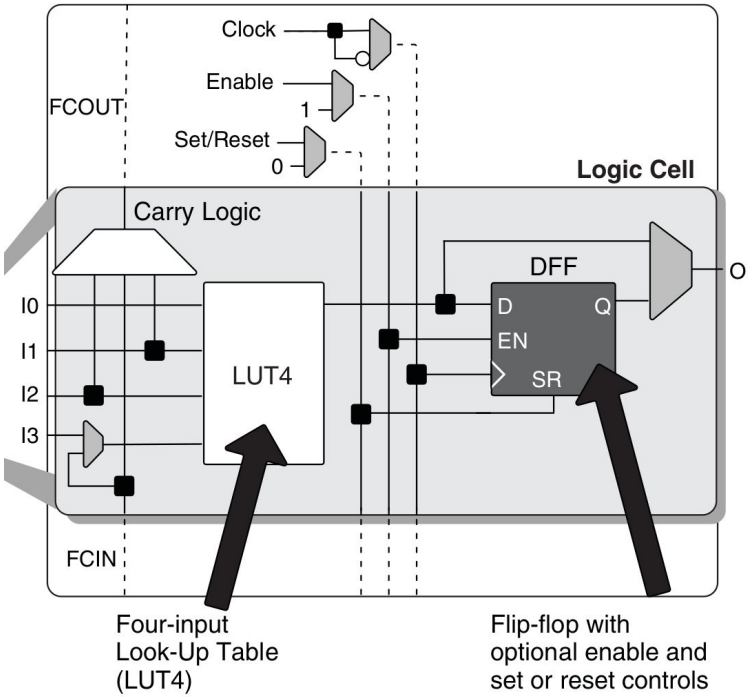
- Output depends on current and previous inputs (state)
- Global clock signal
- State only changes on clock edges (typically only positive edges)
- Example: D-type flip flop
  - (Implemented with logic gates whose outputs loop back to their own inputs)



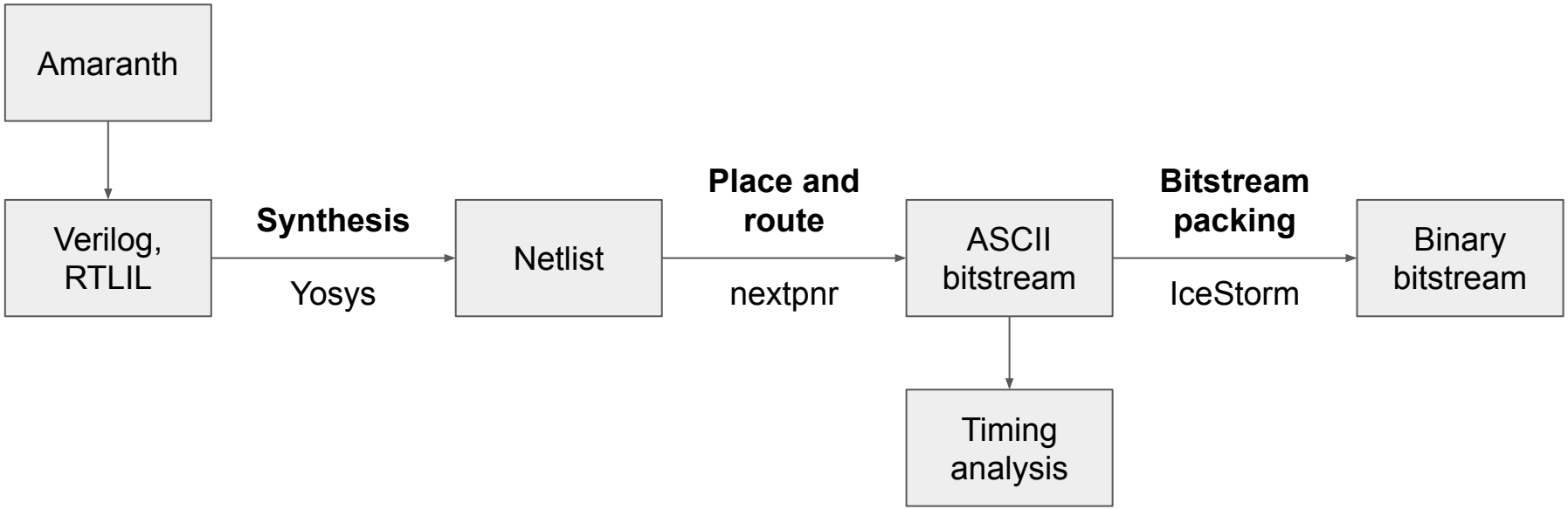


FPGA architecture

- **Logic cells**
  - Look-up table (LUT)
  - D-type flip flop (DFF)
- **Interconnect**
- **Hard blocks - examples:**
  - Block RAM
  - Adder
  - Multiplier
  - Phase-locked loop (PLL)
  - I/O

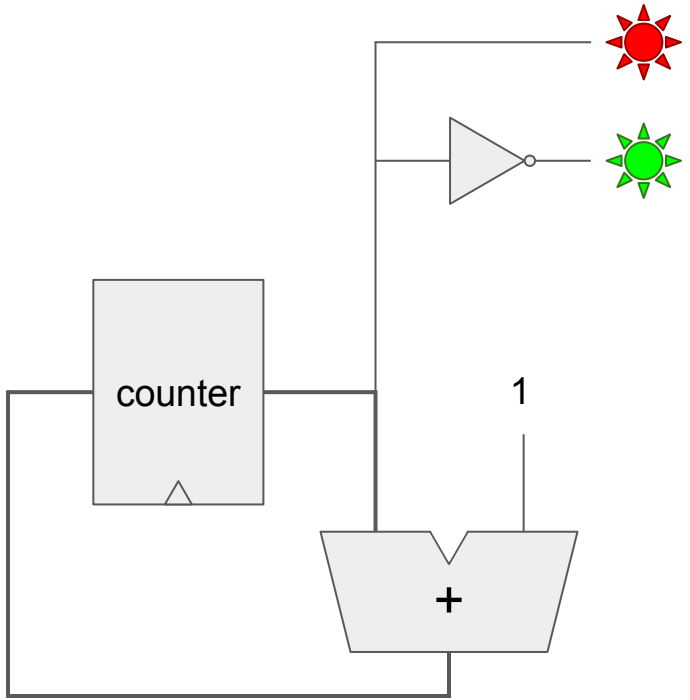


Open-source FPGA toolchain

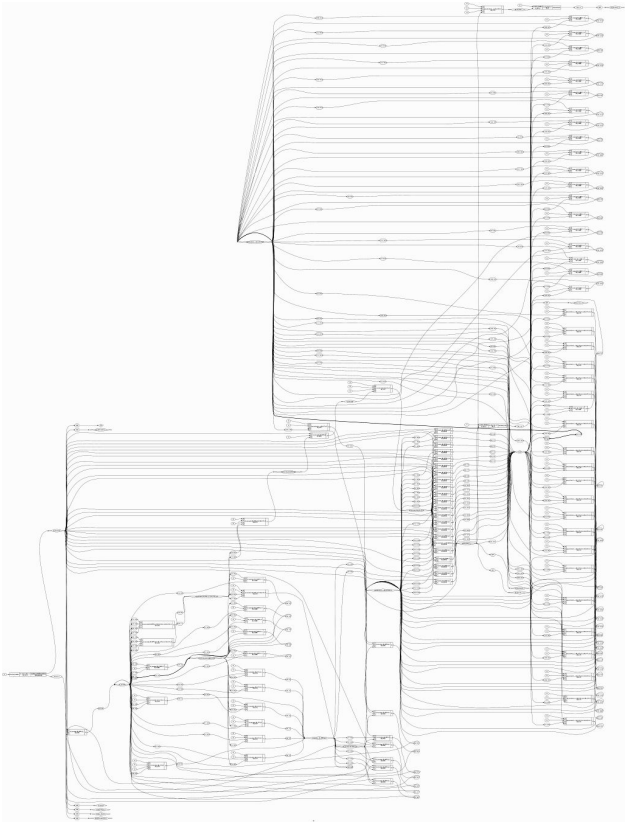


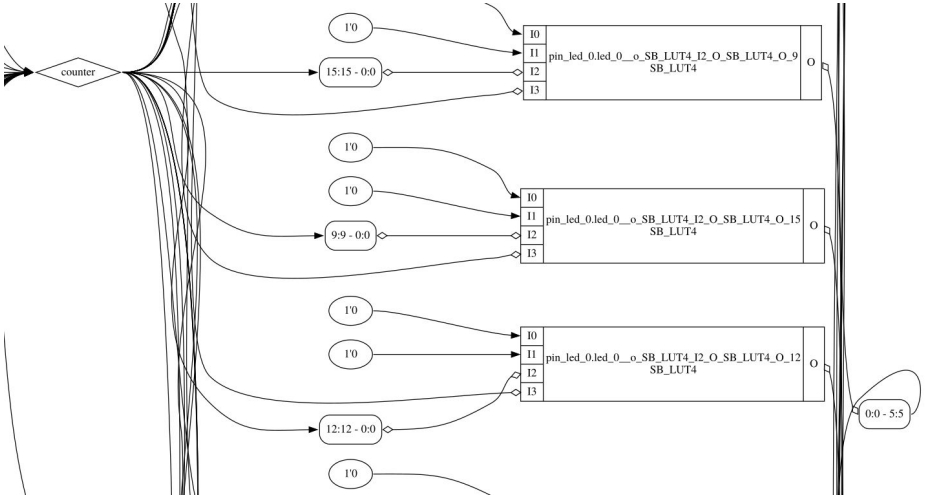
Hardware description language

```python
#!/usr/bin/env python3

from amaranth import *
from amaranth_boards.icebreaker import ICEBreakerPlatform


class Blinky(Elaboratable):
    def elaborate(self, platform):
        m = Module()

        counter = Signal(22)
        m.d.sync += counter.eq(counter + 1)

        led_r = platform.request("led", 0)
        m.d.comb += led_r.eq(counter[-1])

        led_g = platform.request("led", 1)
        m.d.comb += led_g.eq(~led_r)

        return m


platform = ICEBreakerPlatform()
platform.build(Blinky(), do_program=True)
```

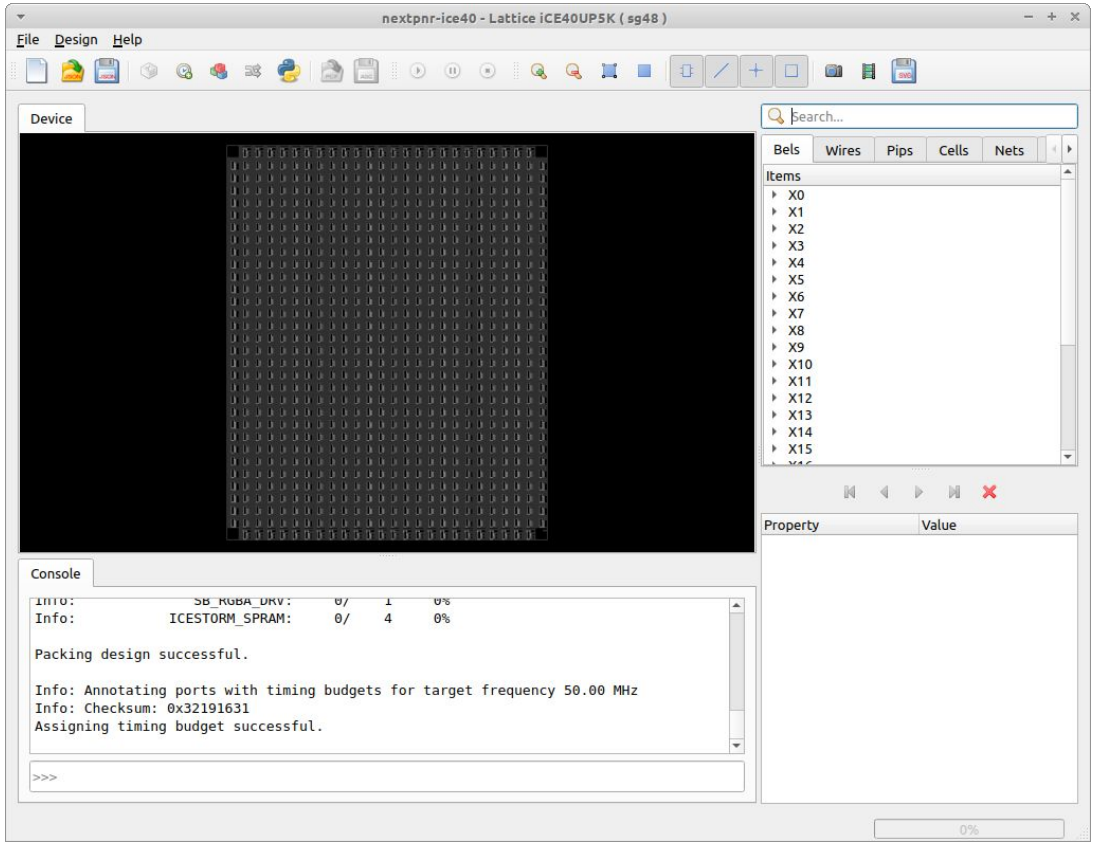


Synthesis

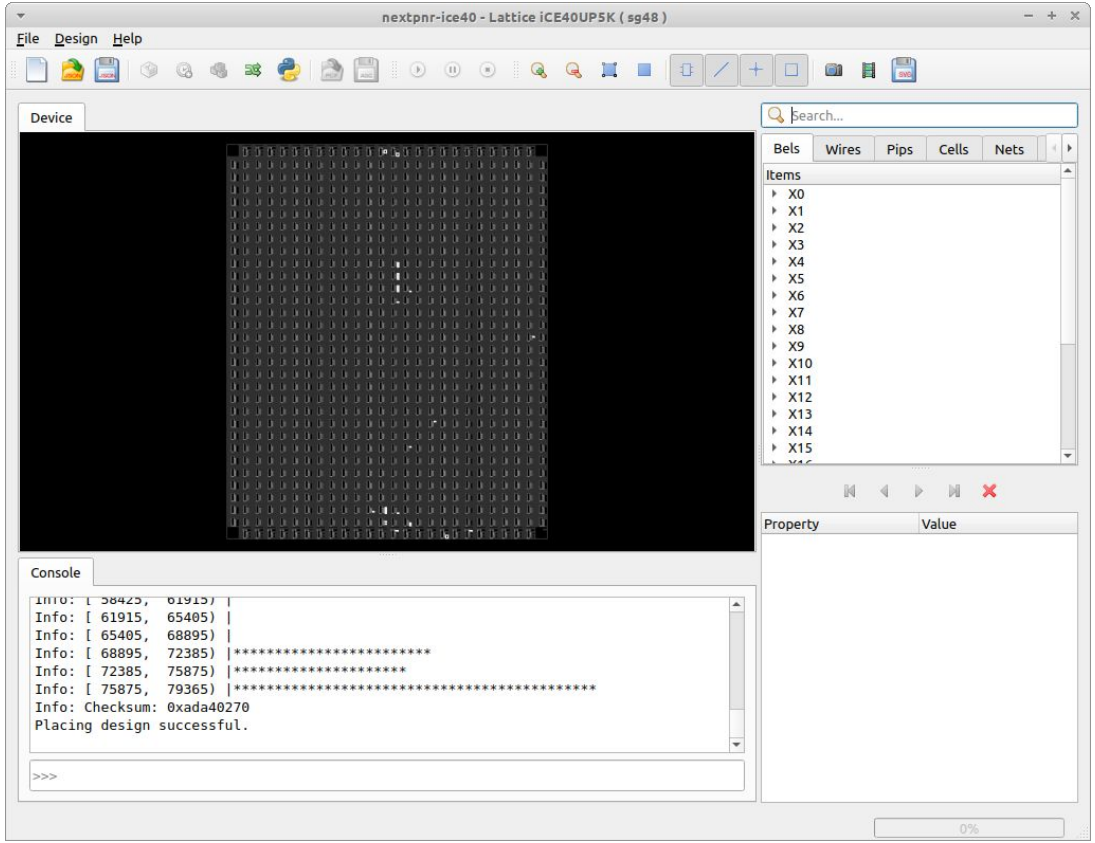



counter
1'0
15:15 - 0:0
I0
I1
I2
I3
pin_led_0.led_0__o_SB_LUT4_I2_O_SB_LUT4_O_9
SB_LUT4
O
1'0
1'0
9:9 - 0:0
I0
I1
I2
I3
pin_led_0.led_0__o_SB_LUT4_I2_O_SB_LUT4_O_15
SB_LUT4
O
1'0
1'0
12:12 - 0:0
1'0
I0
I1
I2
I3
pin_led_0.led_0__o_SB_LUT4_I2_O_SB_LUT4_O_12
SB_LUT4
O
0:0 - 5:5

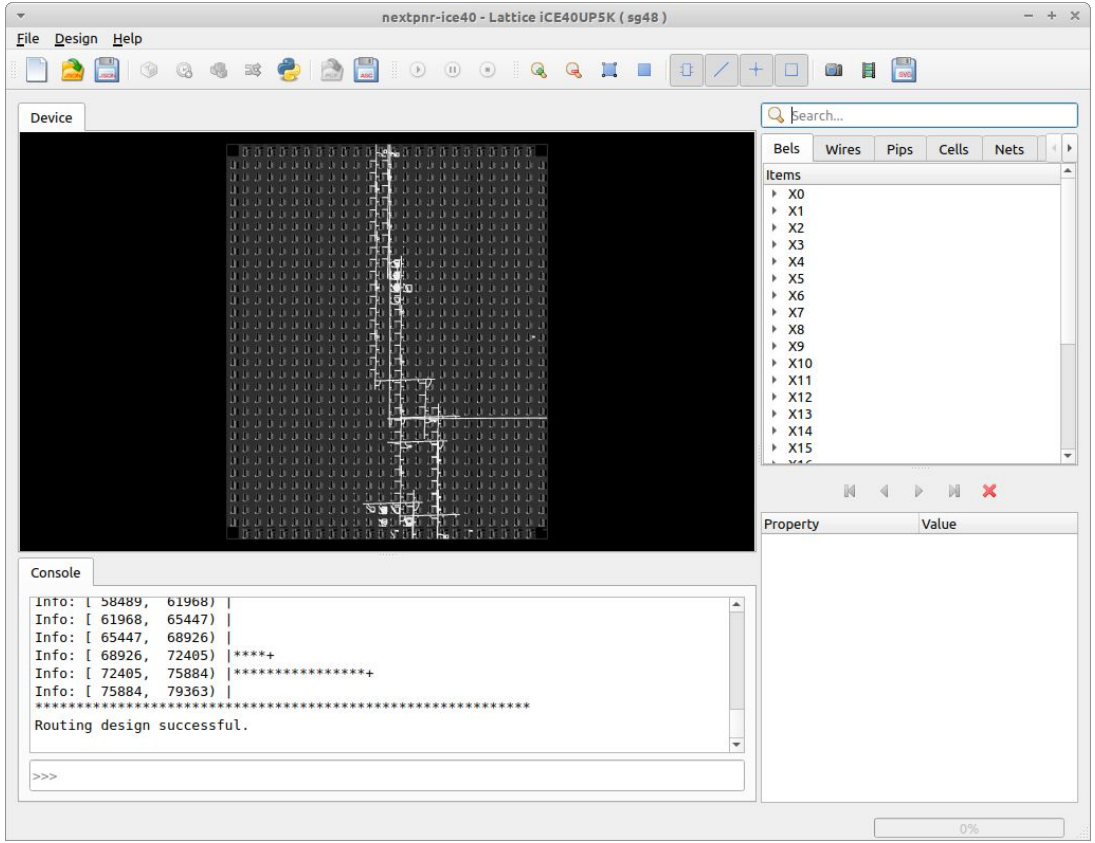
Place and route



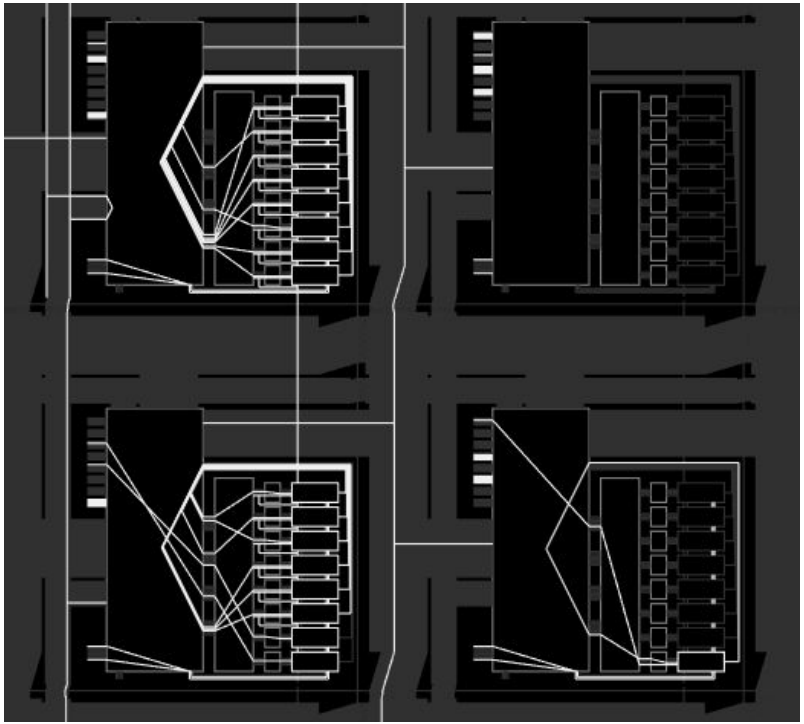
Place and route

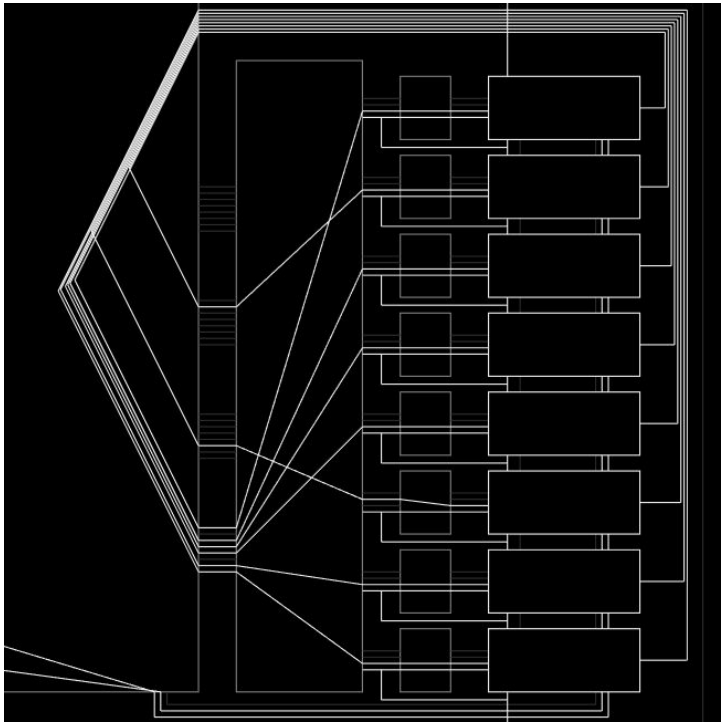


Place and route



Place and route

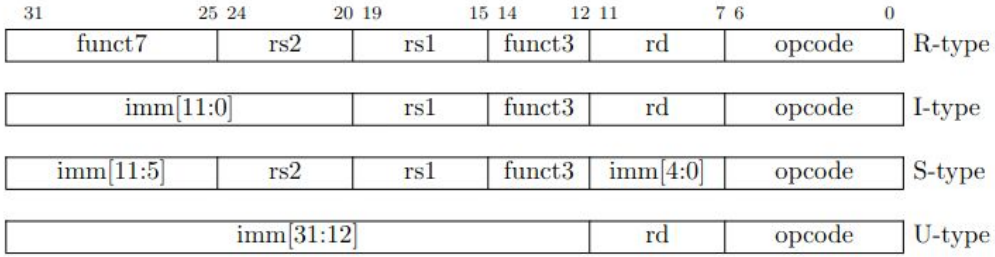




Icicle

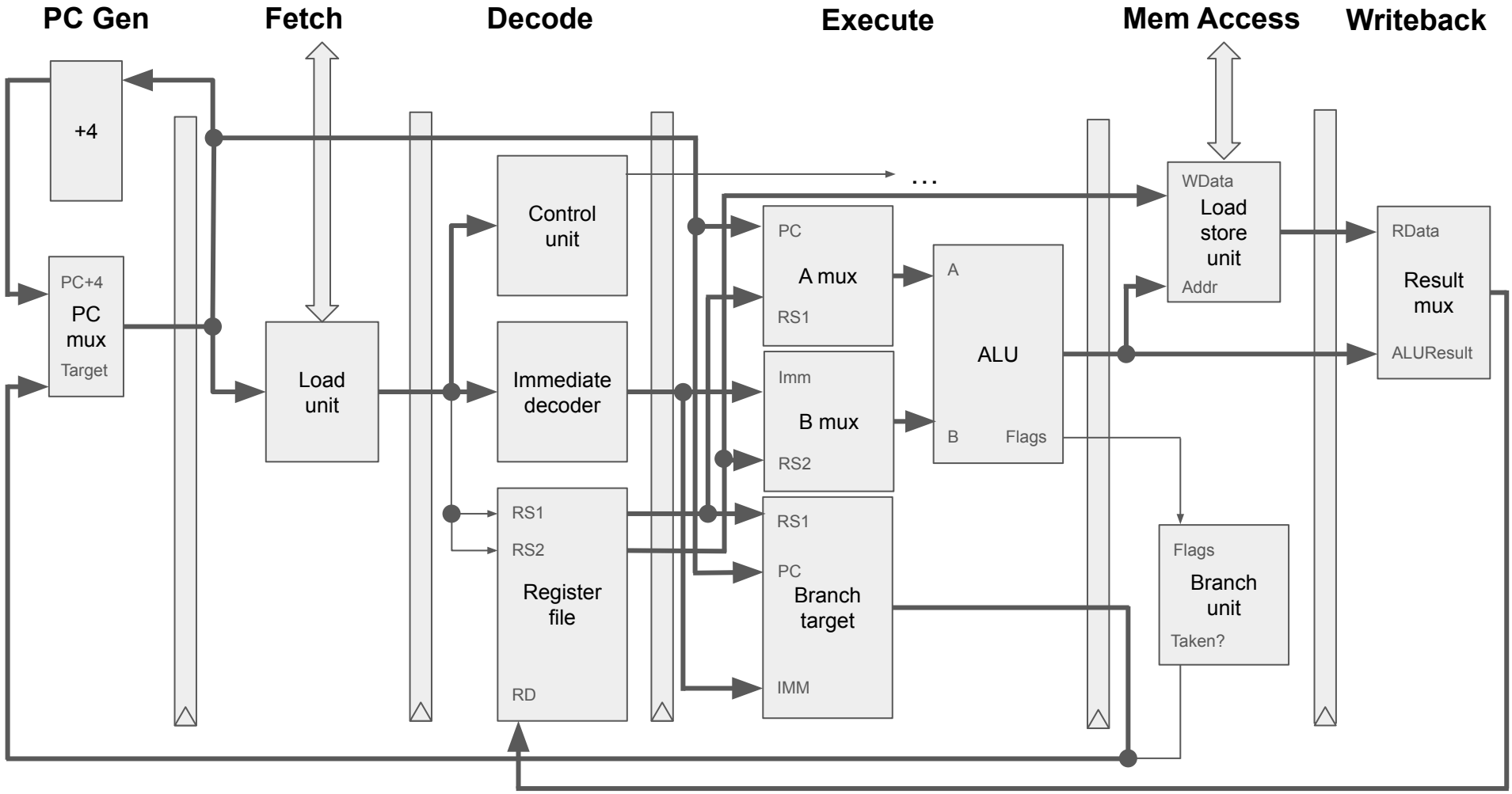
RISC-V

- Open-source instruction set architecture
- Why?
  - Simple
  - Modular
  - Small
    - ~40 instructions in the base ISA
  - Assemblers and compilers already exist



Base Integer Instructions: RV32I, RV

Category	Name	Fmt	RV32I Base
Loads	Load Byte	I	`LB rd,rs1,imm`
	Load Halfword	I	`LH rd,rs1,imm`
	Load Word	I	`LW rd,rs1,imm`
	Load Byte Unsigned	I	`LBU rd,rs1,imm`
	Load Half Unsigned	I	`LHU rd,rs1,imm`
Stores	Store Byte	S	`SB rs1,rs2,imm`
	Store Halfword	S	`SH rs1,rs2,imm`
	Store Word	S	`SW rs1,rs2,imm`
Shifts	Shift Left	R	`SLL rd,rs1,rs2`
	Shift Left Immediate	I	`SLLI rd,rs1,shamt`
	Shift Right	R	`SRL rd,rs1,rs2`
	Shift Right Immediate	I	`SRLI rd,rs1,shamt`
	Shift Right Arithmetic	R	`SRA rd,rs1,rs2`
	Shift Right Arith Imm	I	`SRAI rd,rs1,shamt`
Arithmetic	ADD	R	`ADD rd,rs1,rs2`
	ADD Immediate	I	`ADDI rd,rs1,imm`
	SUBtract	R	`SUB rd,rs1,rs2`
	Load Upper Imm	U	`LUI rd,imm`
	Add Upper Imm to PC	U	`AUIPC rd,imm`
Logical	XOR	R	`XOR rd,rs1,rs2`
	XOR Immediate	I	`XORI rd,rs1,imm`
	OR	R	`OR rd,rs1,rs2`
	OR Immediate	I	`ORI rd,rs1,imm`
	AND	R	`AND rd,rs1,rs2`
	AND Immediate	I	`ANDI rd,rs1,imm`
Compare	Set <	R	`SLT rd,rs1,rs2`
	Set < Immediate	I	`SLTI rd,rs1,imm`
	Set < Unsigned	R	`SLTU rd,rs1,rs2`
	Set < Imm Unsigned	I	`SLTIU rd,rs1,imm`
Branches	Branch =	SB	`BEQ rs1,rs2,imm`
	Branch ≠	SB	`BNE rs1,rs2,imm`
	Branch <	SB	`BLT rs1,rs2,imm`
	Branch ≥	SB	`BGE rs1,rs2,imm`
	Branch < Unsigned	SB	`BLTU rs1,rs2,imm`
	Branch ≥ Unsigned	SB	`BGEU rs1,rs2,imm`
Jump & Link	J&L	UJ	`JAL rd,imm`
	Jump & Link Register	UJ	`JALR rd,rs1,imm`
Synch	Synch thread	I	`FENCE`
	Synch Instr & Data	I	`FENCE.I`
System	System CALL	I	`SCALL`
	System BREAK	I	`SBREAK`
Counters	ReaD CYCLE	I	`RDCYCLE rd`
	ReaD CYCLE upper Half	I	`RDCYCLEH rd`
	ReaD TIME	I	`RDTIME rd`
	ReaD TIME upper Half	I	`RDTIMEH rd`
	ReaD INSTR RETired	I	`RDINSTRET rd`
	ReaD INSTR upper Half	I	`RDINSTRETH rd`


PC Gen
Fetch
Decode
Execute
Mem Access
Writeback
+4
PC+4
PC mux
Target
Load unit
Control unit
Immediate decoder
RS1
RS2
Register file
RD
...
PC
A mux
RS1
Imm
B mux
RS2
RS1
PC
Branch target
IMM
A
ALU
B
Flags
WData
Load store unit
Addr
Flags
Branch unit
Taken?
RData
Result mux
ALUResult

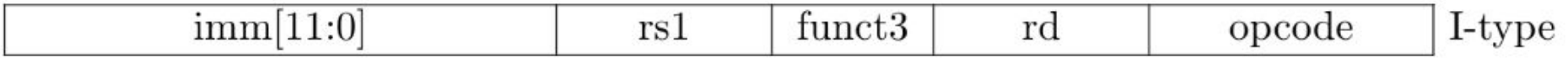
PC Generation

```
class PCGen(Stage):
    def __init__(self, reset_vector=0):
        super().__init__(o_layout=PF_LAYOUT)
        self.o.pc_rdata.reset = reset_vector - 4
        self.branch_taken = Signal()
        self.branch_target = Signal(32)

    def elaborate_stage(self, m, platform):
        with m.If(self.branch_taken):
            m.d.sync += self.o.pc_rdata.eq(self.branch_target)
        with m.Elif(~self.stall):
            m.d.sync += self.o.pc_rdata.eq(self.o.pc_rdata + 4)
```

Fetch

```
addi x5, x6, 1
```

```
00000000000100110000001010010011
```




```
imm=1 rs1=6 funct3=ADDI rd=5 opcode=OP_IMM
```

Decode

```
with m.Switch(opcode):
    with m.Case(Opcode.LUI):
        m.d.comb += self.fmt.eq(Format.U)
    with m.Case(Opcode.AUIPC):
        m.d.comb += self.fmt.eq(Format.U)
    with m.Case(Opcode.JAL):
        m.d.comb += self.fmt.eq(Format.J)
    with m.Case(Opcode.JALR):
        m.d.comb += self.fmt.eq(Format.I)
    with m.Case(Opcode.BRANCH):
        m.d.comb += self.fmt.eq(Format.B)
    with m.Case(Opcode.LOAD):
        m.d.comb += self.fmt.eq(Format.I)
    with m.Case(Opcode.STORE):
        m.d.comb += self.fmt.eq(Format.S)
    with m.Case(Opcode.OP_IMM):
        m.d.comb += self.fmt.eq(Format.I)
    with m.Case(Opcode.OP):
        m.d.comb += self.fmt.eq(Format.R)
    ...
```

Decode

```
m.d.comb += [
    self.rd_wen.eq(self.rd.bool() & self.fmt.matches(Format.R, Format.I, Format.U, Format.J)),
    self.rs1_ren.eq(self.rs1.bool() & self.fmt.matches(Format.R, Format.I, Format.S, Format.B)),
    self.rs2_ren.eq(self.rs2.bool() & self.fmt.matches(Format.R, Format.S, Format.B))
]
```

Decode

```
class RegisterFile(Elaboratable):
    def __init__(self):
        self.rs1_port = Record(RS_PORT_LAYOUT)
        self.rs2_port = Record(RS_PORT_LAYOUT)
        self.rd_port = Record(RD_PORT_LAYOUT)

    def elaborate(self, platform):
        m = Module()

        regs = Memory(width=32, depth=32)
        rs1_port = m.submodules.rs1_port = regs.read_port(transparent=False)
        rs2_port = m.submodules.rs2_port = regs.read_port(transparent=False)
        rd_port = m.submodules.rd_port = regs.write_port()

        m.d.comb += ...

        return m
```

Decode

```
m.d.comb += [
    sign.eq(self.insn[31]),
    imm_i.eq(Cat(self.insn[20], self.insn[21:25], self.insn[25:31], Repl(sign, 21))),
    imm_s.eq(Cat(self.insn[7], self.insn[8:12], self.insn[25:31], Repl(sign, 21))),
    imm_b.eq(Cat(C(0, 1), self.insn[8:12], self.insn[25:31], self.insn[7], Repl(sign, 20))),
    imm_u.eq(Cat(C(0, 12), self.insn[12:20], self.insn[20:31], sign)),
    imm_j.eq(Cat(C(0, 1), self.insn[21:25], self.insn[25:31], self.insn[20], self.insn[12:20], Repl(sign, 12)))
]

with m.Switch(self.fmt):
    with m.Case(Format.I):
        m.d.comb += self.imm.eq(imm_i)
    with m.Case(Format.S):
        m.d.comb += self.imm.eq(imm_s)
    with m.Case(Format.B):
        m.d.comb += self.imm.eq(imm_b)
    with m.Case(Format.U):
        m.d.comb += self.imm.eq(imm_u)
    with m.Case(Format.J):
        m.d.comb += self.imm.eq(imm_j)
```

Decode

```
with m.Switch(opcode):
    with m.Case(Opcode.OP_IMM, Opcode.OP):
        m.d.comb += [
            self.a_sel.eq(ASel.RS1),
            self.b_sel.eq(Mux(opcode == Opcode.OP, BSel.RS2, BSel.IMM)),
            self.wdata_sel.eq(WDataSel.ALU_RESULT),
        ]

        with m.Switch(funct3):
            with m.Case(Funct3.ADD_SUB):
                m.d.comb += [
                    self.result_sel.eq(ResultSel.ADDER),
                    self.add_sub.eq(Mux(opcode == Opcode.OP, funct7[5], 0)),
                ]
            ...
    ...
```

Execute

```
class OperandMux(Elaboratable):
    def __init__(self):
        self.a_sel = Signal(ASel)
        self.b_sel = Signal(BSel)
        self.pc_rdata = Signal(32)
        self.rs1_rdata = Signal(32)
        self.rs2_rdata = Signal(32)
        self.imm = Signal(32)
        self.a = Signal(32)
        self.b = Signal(32)

    def elaborate(self, platform):
        m = Module()

        with m.Switch(self.a_sel):
            with m.Case(ASel.RS1):
                m.d.comb += self.a.eq(self.rs1_rdata)
            with m.Case(ASel.PC):
                m.d.comb += self.a.eq(self.pc_rdata)

        with m.Switch(self.b_sel):
            with m.Case(BSel.RS2):
                m.d.comb += self.b.eq(self.rs2_rdata)
            with m.Case(BSel.IMM):
                m.d.comb += self.b.eq(self.imm)

        return m
```

Execute

```
class Adder(Elaboratable):
    def __init__(self):
        self.sub = Signal()
        self.a = Signal(32)
        self.b = Signal(32)
        self.result = Signal(32)
        self.carry = Signal()

    def elaborate(self, platform):
        m = Module()
        m.d.comb += Cat(self.result, self.carry).eq(Mux(self.sub, self.a - self.b, self.a + self.b))
        return m
```

Memory Access

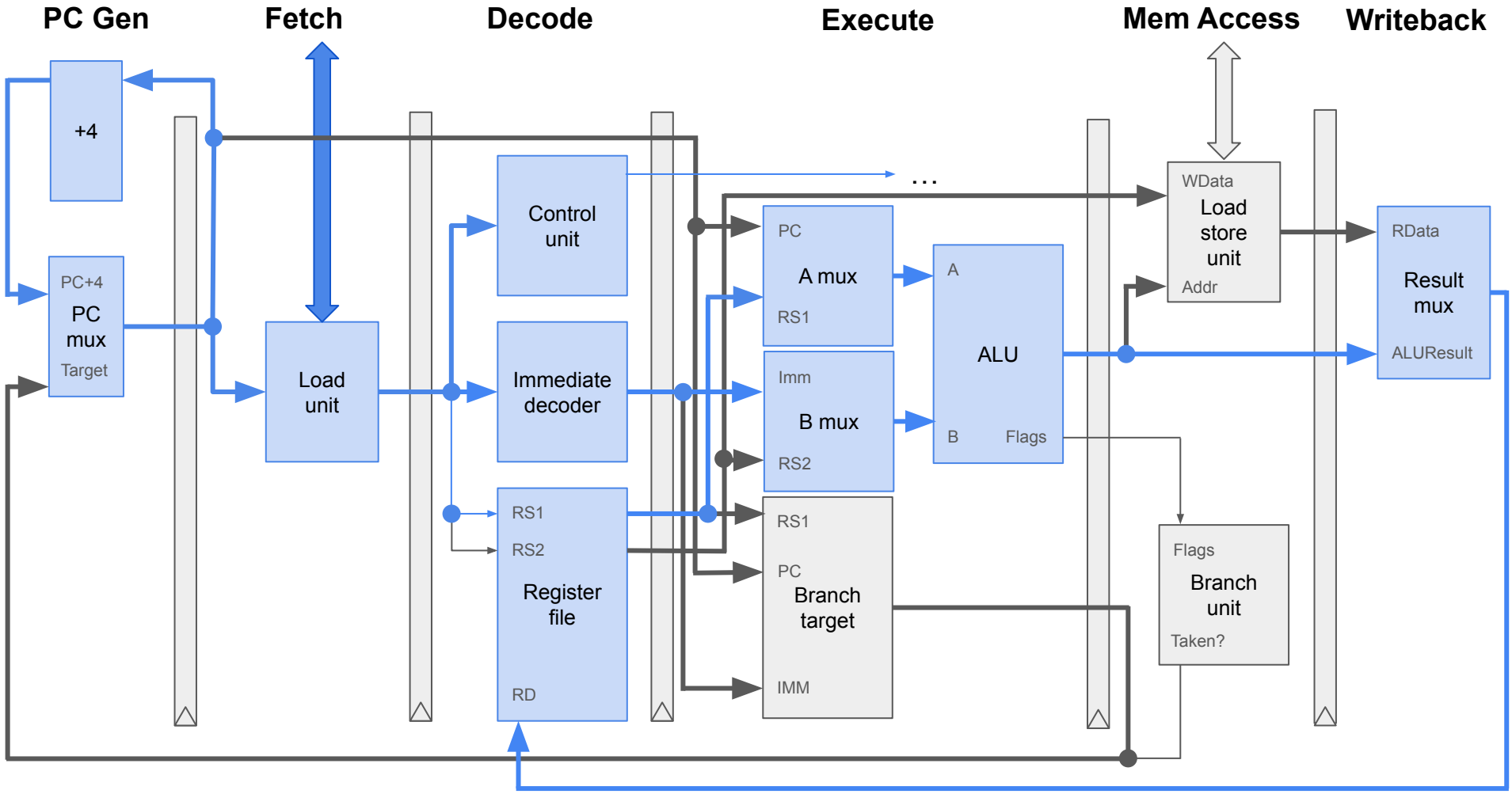
Nothing interesting happens.

Writeback

```
class WDataMux(Elaboratable):
    def __init__(self):
        self.sel = Signal(WDataSel)
        self.result = Signal(32)
        self.mem_rdata = Signal(32)
        self.rd_wdata = Signal(32)

    def elaborate(self, platform):
        m = Module()

        with m.Switch(self.sel):
            with m.Case(WDataSel.ALU_RESULT):
                m.d.comb += self.rd_wdata.eq(self.result)
            with m.Case(WDataSel.MEM_RDATA):
                m.d.comb += self.rd_wdata.eq(self.mem_rdata)

        return m
```

Writeback

```
class Writeback(Stage):
    def __init__(self):
        super().__init__(i_layout=MW_LAYOUT)
        self.rd_port = Record(RD_PORT_LAYOUT)

    def elaborate_stage(self, m, platform):
        wdata_mux = m.submodules.wdata_mux = WDataMux()
        m.d.comb += [
            wdata_mux.sel.eq(self.i.wdata_sel),
            wdata_mux.result.eq(self.i.result),
            wdata_mux.mem_rdata.eq(self.i.mem_rdata)
        ]

        m.d.comb += [
            self.rd_port.en.eq(~self.stall & (self.i.state == State.VALID) & self.i.rd_wen),
            self.rd_port.addr.eq(self.i.rd),
            self.rd_port.data.eq(wdata_mux.rd_wdata)
        ]
```

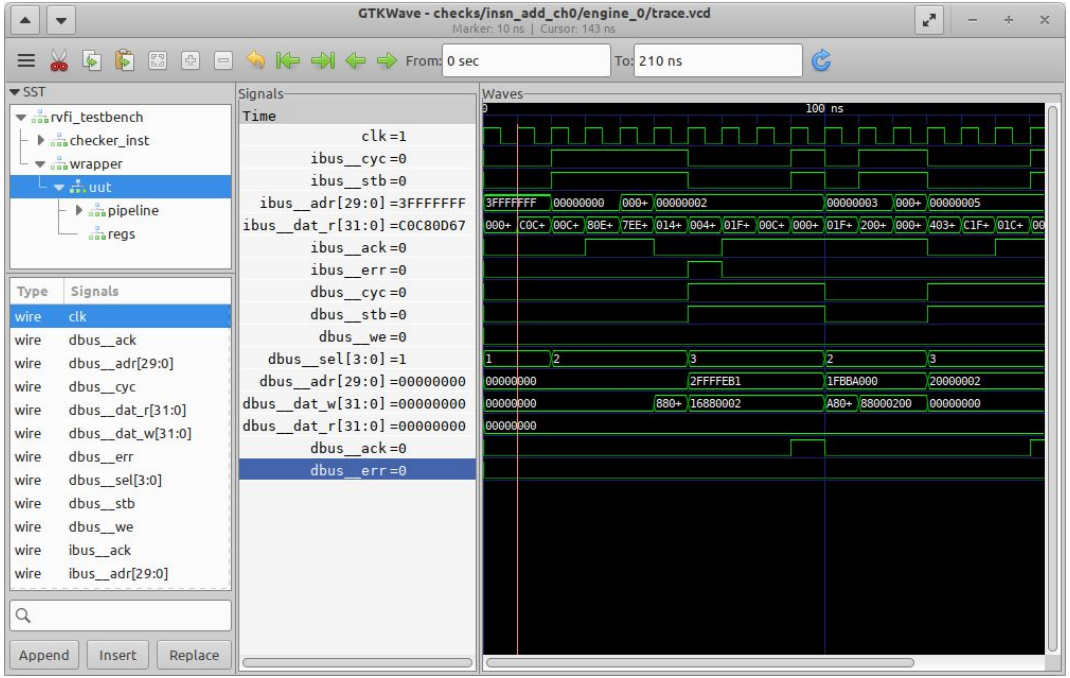

PC Gen
Fetch
Decode
Execute
Mem Access
Writeback
+4
PC+4
PC mux
Target
Load unit
Control unit
Immediate decoder
RS1
RS2
Register file
RD
...
PC
A mux
RS1
Imm
B mux
RS2
RS1
PC
Branch target
IMM
A
ALU
B
Flags
WData
Load store unit
Addr
Flags
Branch unit
Taken?
RData
Result mux
ALUResult

What we didn’t cover

- Load/store units
  - Address, write data out
  - Read data in
  - Valid/ready handshake
- Branch unit
  - Target calculation
  - Predict not taken - PC+4
  - Flush partially executed instructions if incorrect
- Hazards
  - RS1/RS2 depend on instruction still being executed
  - Solutions:
    - Bypassing
    - Interlocking

Simulation

```
class AdderTestCase(TestCase):
    def test_add(self):
        m = Adder()
        sim = Simulator(m)

        def process():
            yield m.sub.eq(0)
            yield m.a.eq(1)
            yield m.b.eq(2)

            yield Delay()

            self.assertEqual((yield m.result), 3)

        sim.add_process(process)
        sim.run()
```



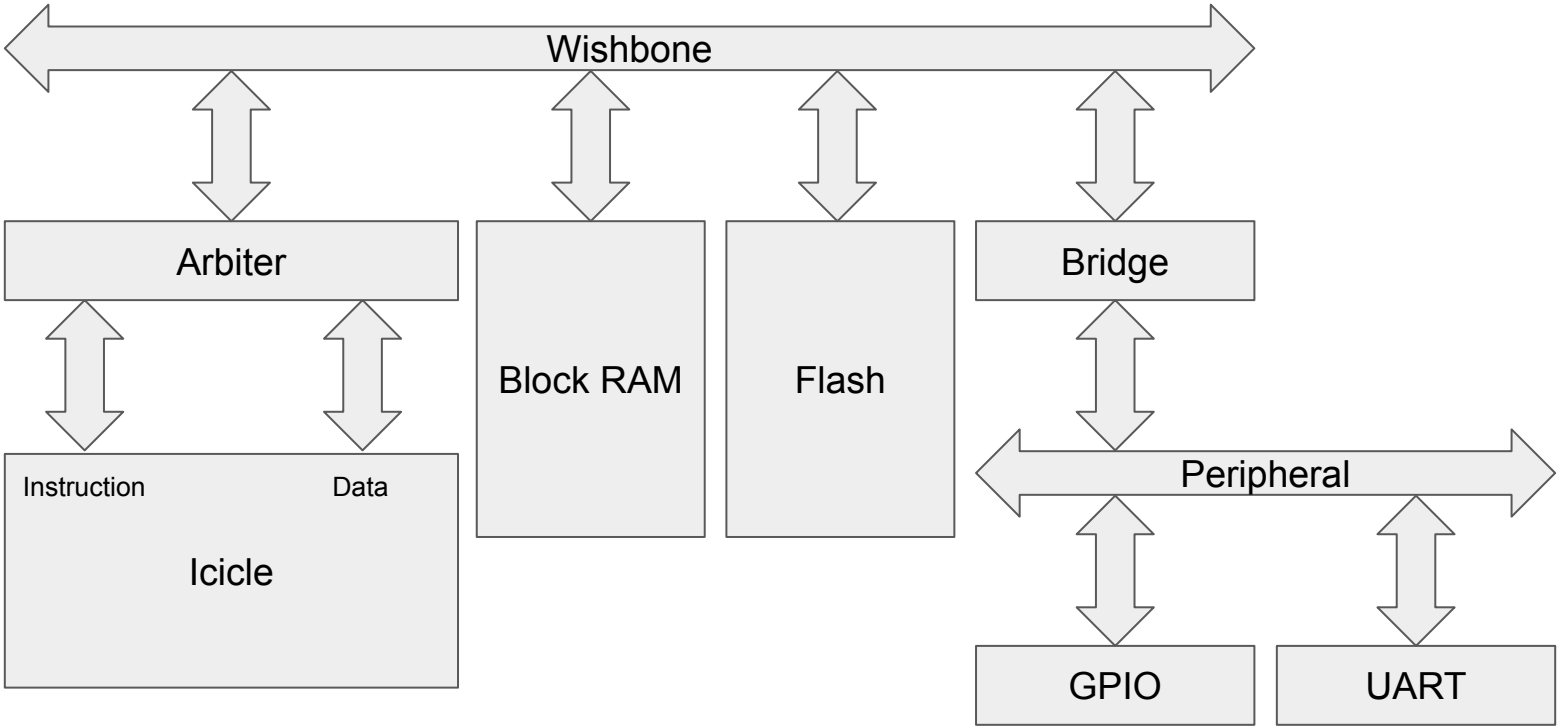
Formal verification

1. Convert circuit to (gigantic) set of mathematical formulae
2. Write specification to verify against (riscv-formal)
3. Compare implementation to specification, output true if they do not match
4. Run SMT solver (fancy SAT solver)
5. If SAT solver finds a solution, circuit is invalid

```
SBY 22:57:12 [insn_sh_ch0] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:25 (25)
SBY 22:57:12 [insn_sh_ch0] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:24 (24)
SBY 22:57:12 [insn_sh_ch0] summary: engine_0 (smtbmc boolector) returned pass
SBY 22:57:12 [insn_sh_ch0] DONE (PASS, rc=0)
SBY 22:57:16 [insn_sw_ch0] engine_0: ##   0:00:22  Status: passed
SBY 22:57:16 [insn_sw_ch0] engine_0: finished (returncode=0)
SBY 22:57:16 [insn_sw_ch0] engine_0: Status returned by engine: pass
SBY 22:57:16 [insn_sw_ch0] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:25 (25)
SBY 22:57:16 [insn_sw_ch0] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:24 (24)
SBY 22:57:16 [insn_sw_ch0] summary: engine_0 (smtbmc boolector) returned pass
SBY 22:57:16 [insn_sw_ch0] DONE (PASS, rc=0)
SBY 22:57:45 [pc_fwd_ch0] engine_0: ##   0:01:01  waiting for solver (1 minute)
SBY 22:57:46 [reg_ch0] engine_0: ##   0:01:02  waiting for solver (1 minute)
SBY 23:00:13 [reg_ch0] engine_0: ##   0:03:30  Status: passed
SBY 23:00:13 [reg_ch0] engine_0: finished (returncode=0)
SBY 23:00:13 [reg_ch0] engine_0: Status returned by engine: pass
SBY 23:00:13 [reg_ch0] summary: Elapsed clock time [H:MM:SS (secs)]: 0:03:32 (212)
SBY 23:00:13 [reg_ch0] summary: Elapsed process time [H:MM:SS (secs)]: 0:03:31 (211)
SBY 23:00:13 [reg_ch0] summary: engine_0 (smtbmc boolector) returned pass
SBY 23:00:13 [reg_ch0] DONE (PASS, rc=0)
```

```
  1[|||||||100.0%]   9[|||||||100.0%]  17[||||||||100.0%] 25[||||||||98.9%]
  2[|||||||100.0%]  10[|||||||100.0%]  18[||||||||100.0%] 26[||||||||100.0%]
  3[||||||||100.0%]  11[|||||||100.0%]  19[||||||||100.0%] 27[||||||||99.4%]
  4[|||||||100.0%]  12[|||||||100.0%]  20[||||||||98.3%] 28[||||||||100.0%]
  5[||||||||100.0%]  13[||||||||100.0%]  21[||||||||100.0%] 29[||||||||100.0%]
  6[||||||||100.0%]  14[|||||||100.0%]  22[||||||||100.0%] 30[||||||||100.0%]
  7[||||||||100.0%]  15[|||||||100.0%]  23[||||||||100.0%] 31[||||||||100.0%]
  8[||||||||100.0%]  16[|||||||100.0%]  24[||||||||100.0%] 32[||||||||99.4%]
Mem[||||||||||||||||||||19.4G/31.3G] Tasks: 239, 1769 thr, 367 kthr; 32 run
Swp[                          0K/0K] Load average: 4.49 1.77 1.56
                                     Uptime: 9 days, 01:15:17
```

System on chip



Freestanding C

```
ENTRY(start)

MEMORY {
   flash (rx) : ORIGIN = 0x00100000, LENGTH = 15M
   ram  (rwx) : ORIGIN = 0x40000000, LENGTH = 128K
}

SECTIONS {
   .text : {
       start.o(.text);
       *(.text);
       *(.text.*);
   } > flash

   .data : ALIGN(4) {
       *(.data);
       *(.data.*);

       . = ALIGN(4);
   } > ram AT> flash

   .rodata : {
       *(.rodata);
       *(.rodata.*);
   } > flash

   .bss : ALIGN(4) {
       *(.bss);
       *(.bss.*);

       . = ALIGN(4);
   } > ram

   bss_start = ADDR(.bss);
   bss_end = bss_start + SIZEOF(.bss);

   data_flash_start = LOADADDR(.data);
   data_start = ADDR(.data);
   data_end = data_start + SIZEOF(.data);

   stack_top = ORIGIN(ram) + LENGTH(ram);
}
```

- Memory layout
- Zero BSS section
- Copy writable data from flash to RAM
- Set stack pointer

```
.extern bss_start
.extern bss_end

.extern data_flash_start
.extern data_start
.extern data_end

.extern stack_top

.global start
start:
    la t0, bss_start
    la t1, bss_end

    beq t0, t1, clear_bss_done
clear_bss:
    sw zero, 0(t0)
    addi t0, t0, 4
    bne t0, t1, clear_bss
clear_bss_done:

    la t0, data_flash_start
    la t1, data_start
    la t2, data_end

    beq t1, t2, copy_data_done
copy_data:
    lw t3, 0(t0)
    sw t3, 0(t1)
    addi t0, t0, 4
    addi t1, t1, 4
    bne t1, t2, copy_data
copy_data_done:

    la sp, stack_top

    call main
    j .
```

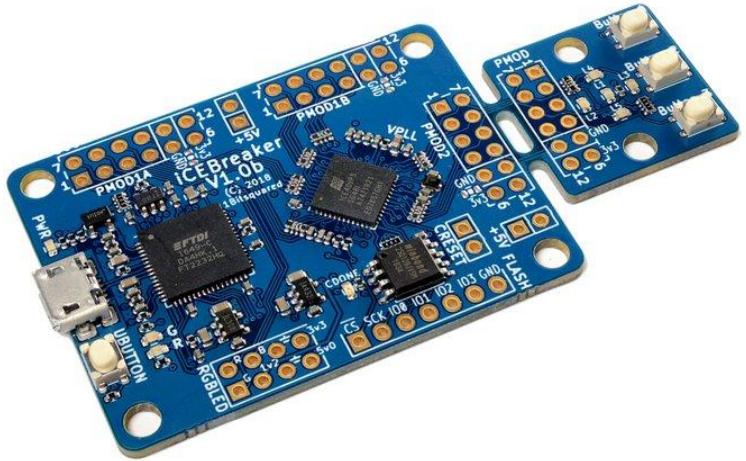
Demo

```c
#include <stdint.h>

static volatile uint8_t *const uart_status = (volatile uint8_t *) 0x80000006;
static volatile uint8_t *const uart_data   = (volatile uint8_t *) 0x80000007;

static const uint8_t UART_STATUS_TX_RDY = 0x2;

static void uart_puts(const char *str) {
    char c;
    while ((c = *str++)) {
        while (!(*uart_status & UART_STATUS_TX_RDY));
        *uart_data = c;
    }
}

int main() {
    for (;;) {
        uart_puts("Hello, world!\r\n");
    }
}
```

```
~$ picocom -b 9600 -q /dev/ttyUSB0
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, wor~$
```

Suggested hardware: iCEBreaker

- Price: $69 (~£55)
- Lattice iCE40UP5K FPGA
  - 5,280 logic cells
  - 8 DSP blocks (for multiply+add)
  - 128 kilobytes of single-port RAM
  - 15 kilobytes of dual-port RAM
- 16 megabytes of flash
- USB <-> serial adapter
- 4 buttons, 1 RGB LED, 7 LEDs
- PMOD connectors for expansion
- Schematics/PCB layout are open-source
- Large enough to implement a 32-bit microcontroller-class CPU + peripherals
- Open-source toolchain



Links

Amaranth (HDL):
https://github.com/amaranth-lang/amaranth

Yosys (synthesis):
https://yosyshq.net/yosys/

nextpnr (place and route):
https://github.com/YosysHQ/nextpnr

Project IceStorm (bitstream generation):
https://clifford.at/icestorm/

RISC-V:
https://riscv.org/specifications/

Icicle:
https://github.com/grahamedgecombe/icicle

riscv-formal:
https://github.com/YosysHQ/riscv-formal

iCEBreaker (development board):
https://github.com/icebreaker-fpga/icebreaker