Projeto de hardware digital com Bluespec

Pequena introdução à lógica digital reconfigurável e à linguagem de descrição de hardware Bluespec

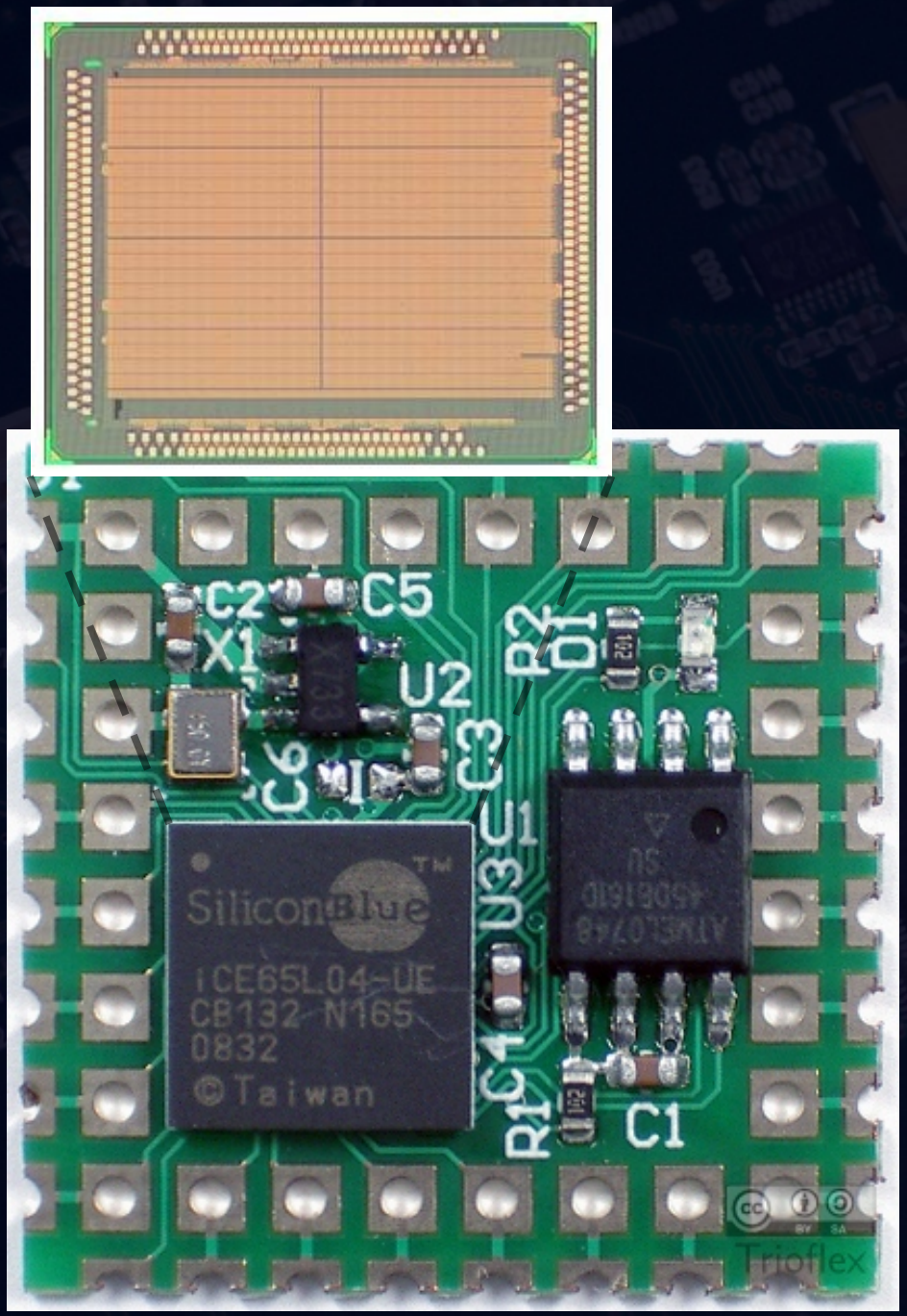
Paulo Matias (IFSC-USP)

ERAD-SP 2013

Por que projetar hardware?

- Máquinas projetadas para propósitos específicos
  - Podem ser mais otimizadas
    que máquinas de propósito geral
  - Não precisam ser programáveis
- Circuitos integrados de propósito específico (ASIC)
  - Alto custo de engenharia não-recorrente (NRE)
  - Impossível corrigir erros de projeto após produção
- Alternativa: lógica reconfigurável (FPGA)
  - Baixo NRE, possibilidade de reprogramação

Estrutura de uma FPGA



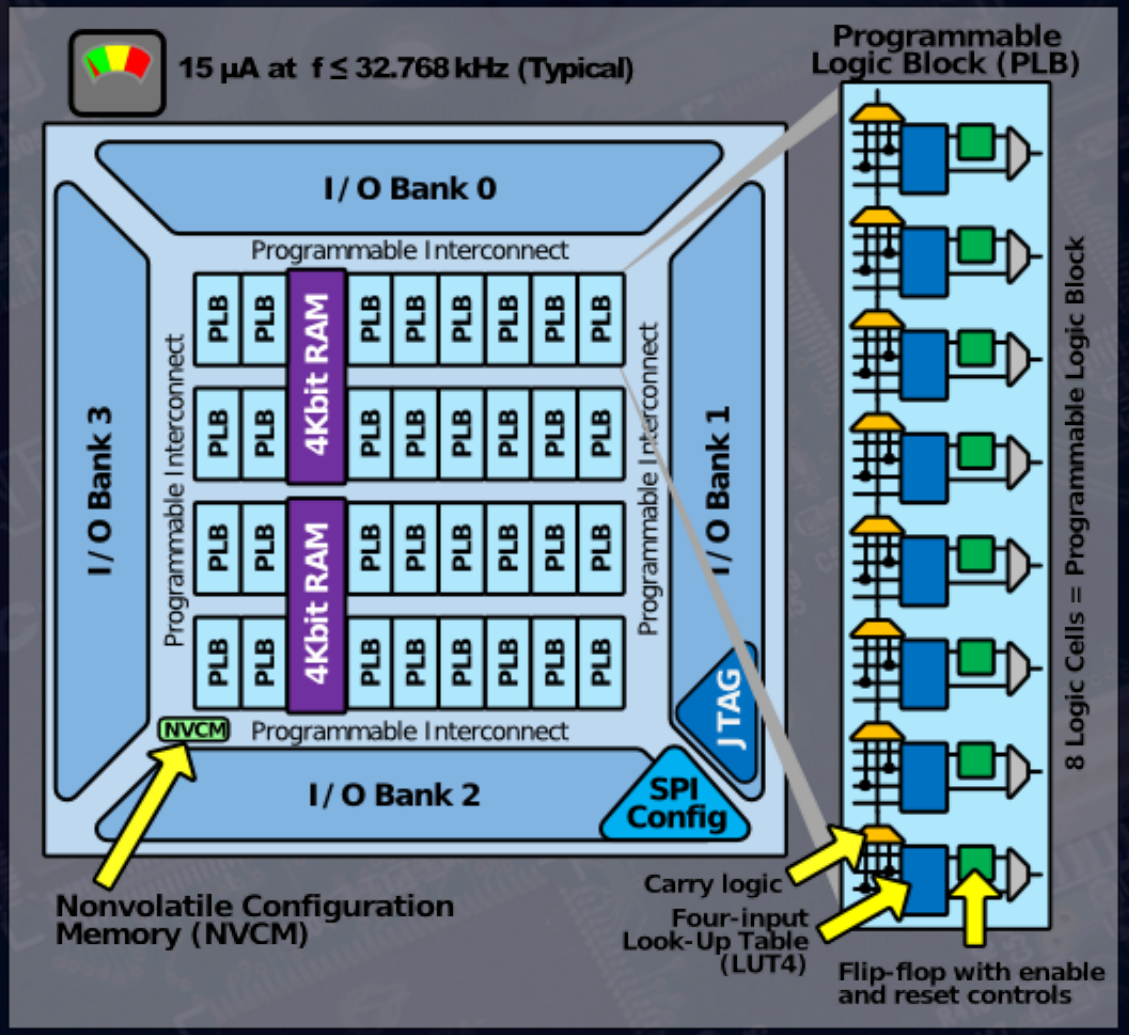
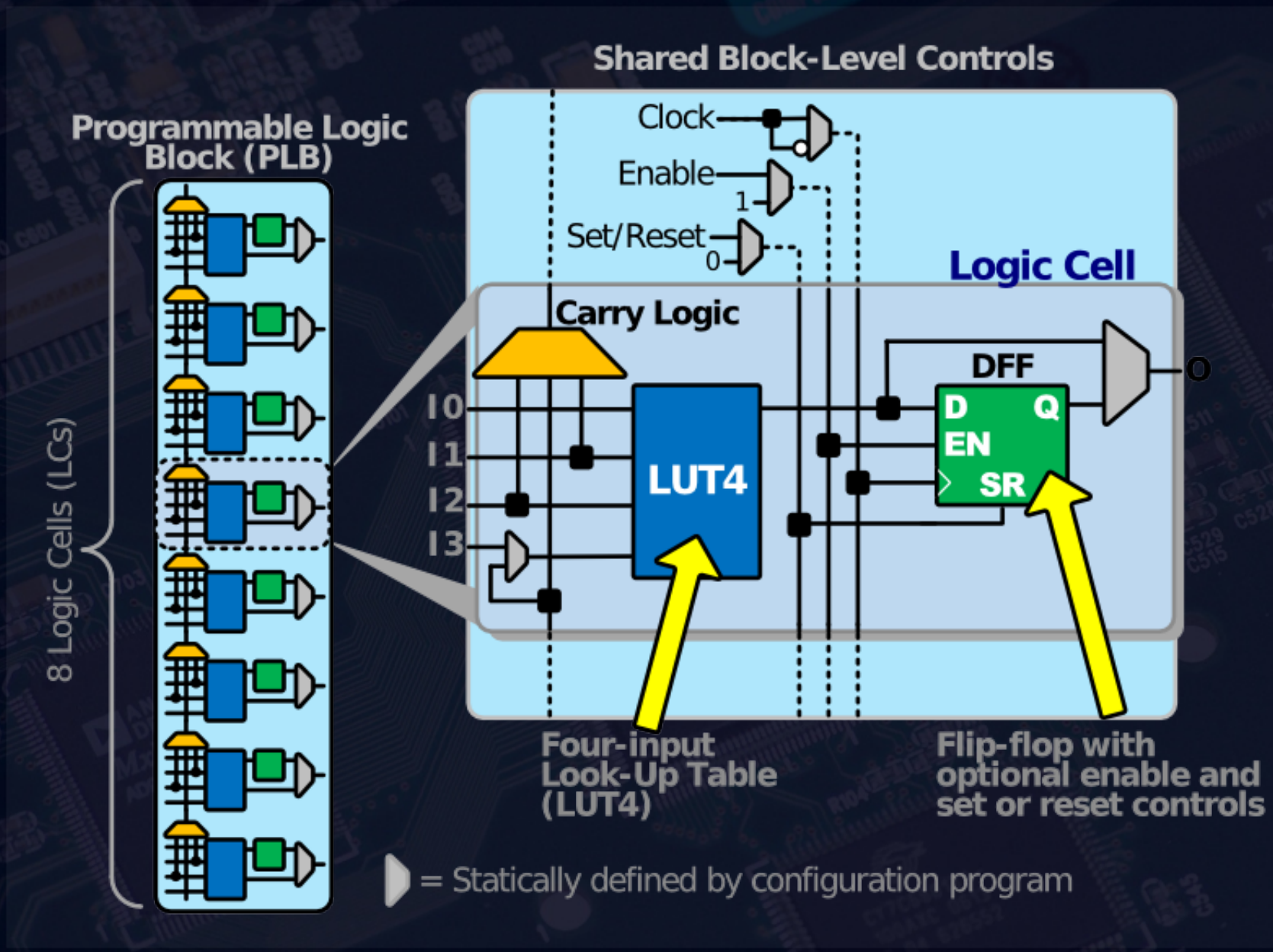


Imagem: Divulgação SiliconBlue

Detalhe de uma célula lógica



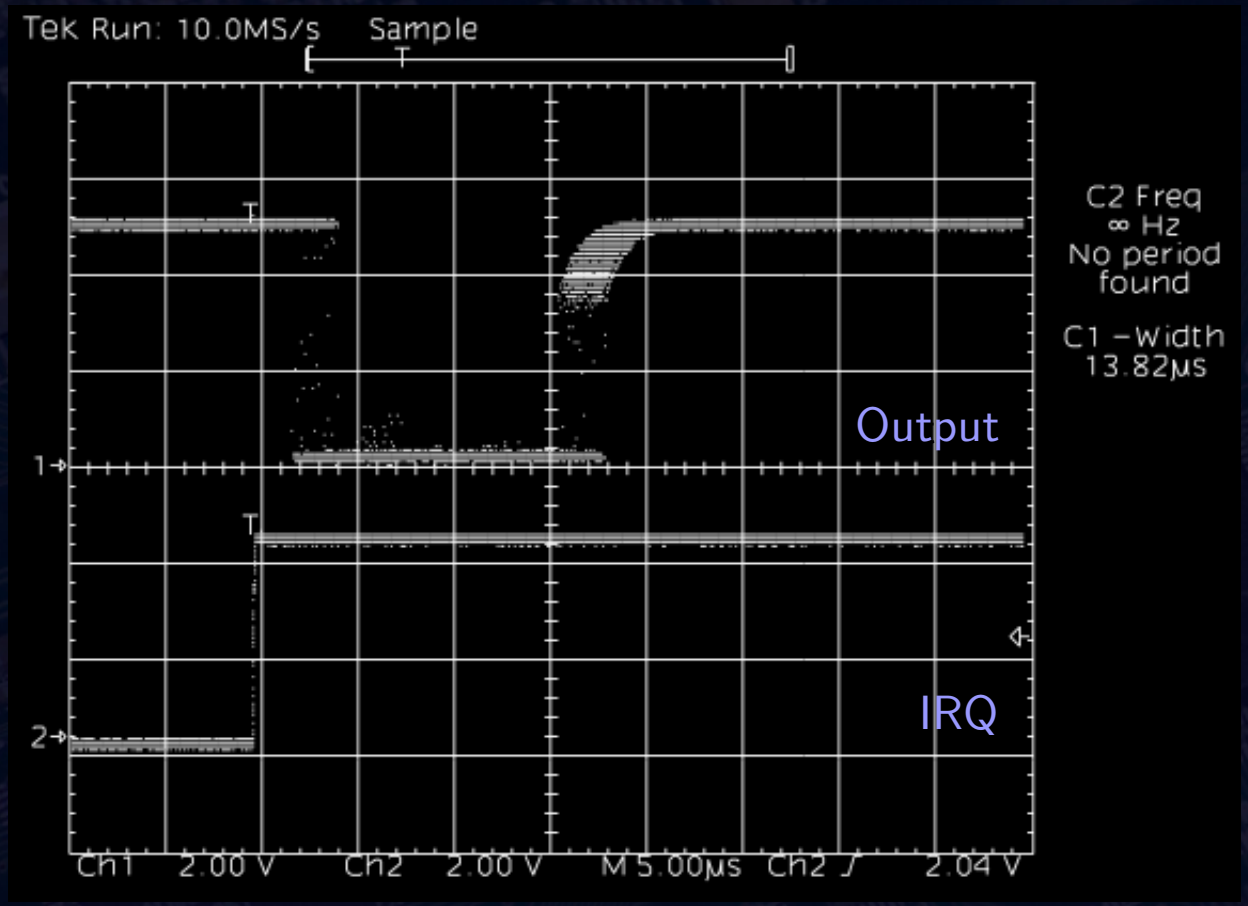
http://www.latticesemi.com/view_document?document_id=44659

Algumas aplicações de FPGAs

- Teste de projetos de circuitos integrados
  - Chiou et. al. ICCAD 2007 – Simulação de arquitetura a **1.2 MIPS** em FPGA com Bluespec vs. **1-200 KIPS** em software.
- Quando é muito caro produzir um ASIC
  - *Glue logic*
  - Computação de alto desempenho com requisitos de:
    - Alto determinismo (*hard real-time*)
    - Baixa latência

Latência e jitter

- RTAI Linux, Athlon 64 X2, baixa carga de CPU



Latência	3 µs
Duração	14 µs
Jitter	3 µs

- Latência da ordem de 20ns e jitter menor que 1ns com uma FPGA de baixo custo a 50 MHz.

Desempenho em cálculos com inteiros

- SHA256: hash (função de via única)
  - Verificação criptográfica de integridade
- Fácil paralelização (inclusive em GPU)
- Utilizado na moeda criptográfica Bitcoin
  - Consolidação do registro de transações
  - Criação de novas moedas (*mineração*)
- Atraiu interesse de diversos públicos
  - Aficionados por criptografia, *Overclockers*, Investidores de risco, Ativistas

SHA256: GPU vs. FPGA vs. ASIC

Produto	Velocidade (Mh/s)	Eficiência custo (Mh/s/$)	Eficiência energia (Mh/J)
ATI Radeon HD6990	**772**	1.23	1.93
ATI Radeon HD5850	346	**3.00**	1.92
ATI Radeon HD7750	123	1.12	**2.46**
Butterflylabs Mini Rig (18x2x Altera Arria II EP2AGX260)	**25 200**	**1.64**	20.16
BitForce Single (2x Altera Stratix III EP3SL150F780)	832	1.38	10.40
FPGA Mining X6500 (2x Xilinx Spartan-6 LX150-3FGG484C)	400	0.72	**23.25**
Avalon ASIC #2	**82 000**	**54.70**	117
BitForce SC 5Gh/s	5000	18.24	**166**
Block Erupter Blade	10752	1.87	129

GPU **FPGA** **ASIC**

https://en.bitcoin.it/wiki/Mining_hardware_comparison

Desempenho em ponto flutuante

- Difícil comparação de *benchmarks* reais
  - Altera: dados disponíveis para matrizes até 400x400
  - GPUs: nessas dimensões, a latência domina
- Comparação com base em valores de pico teóricos

Produto	Velocidade [*] (**GFLOP/s**)	Eficiência custo (**GFLOP/s/$**)	Eficiência energia (**GFLOP/J**)
NVidia Tesla C2050	1030	1.31	5.7
NVidia Tesla K20x	3935	1.04	16
Altera Stratix V GS (5SGSMD8)	1250	0.18	6.2 [#]

[*] Estimativa de pico de *throughput* para operações mul-add em precisão simples.

[#] Medida em um *benchmark* real (Decomposição QR). Demais dados utilizam Pico/TDP.

Achieving One TeraFLOPS with 28-nm FPGAs, Altera Corp., Sep 2010.

Floating-point DSP Energy Efficiency on Altera 28nm FPGAs, Berkeley Design Tech. Inc., Feb 2013.

QPACE: Interconexão customizada

- Operações de ponto flutuante: PowerXCell 8i
- Rede 3D-Torus: FPGA Xilinx Virtex-5 lx110t
  - M. Pivanti, A Scalable Parallel Architecture with FPGA-Based Network Processor for Scientific Computing, Ph.D thesis, 2012.

Data	Posição **Top500**	Posição **Green500**	Eficiência energia (**MFLOP/J**)	Potência (**kW**)
11/2009	110	1	722.98	59.49
06/2010	131	1	773.38	57.54
11/2010	209	5	773.38	57.54
06/2011	406	7	773.38	57.54

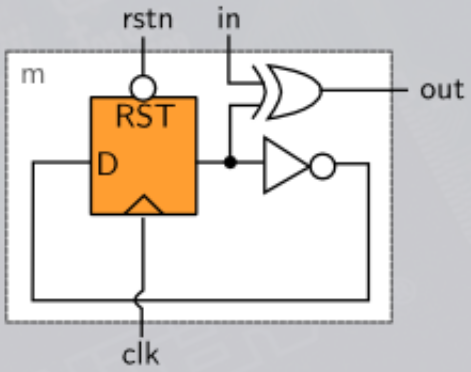
http://green500.org

Fluxo tradicional de projeto com FPGA

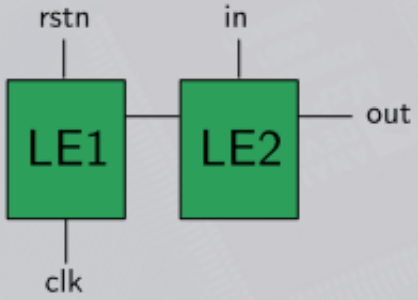
RTL

```
module m(clk, rstn, in, out);
input clk; input rstn; input in;
output out;
always @(negedge rstn or posedge clk)
  if(!rstn)
    ff1 <= 0;
  else
    ff1 <= ~ff1;
  end
assign out = ff1 ^ in;
endmodule
```

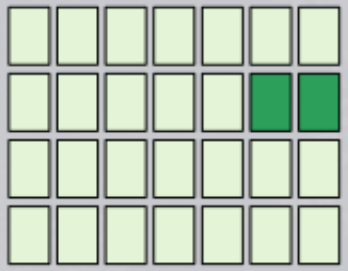
Gate-level



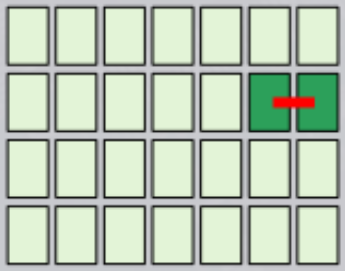
Tech-Map



Place



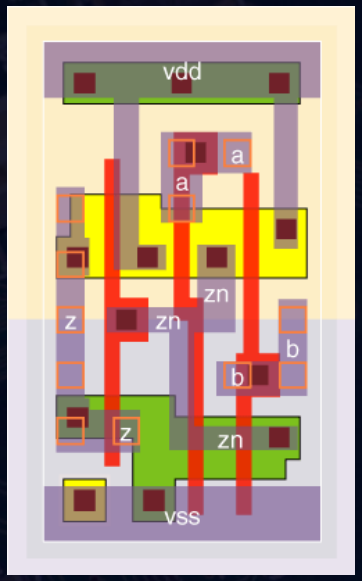
Route



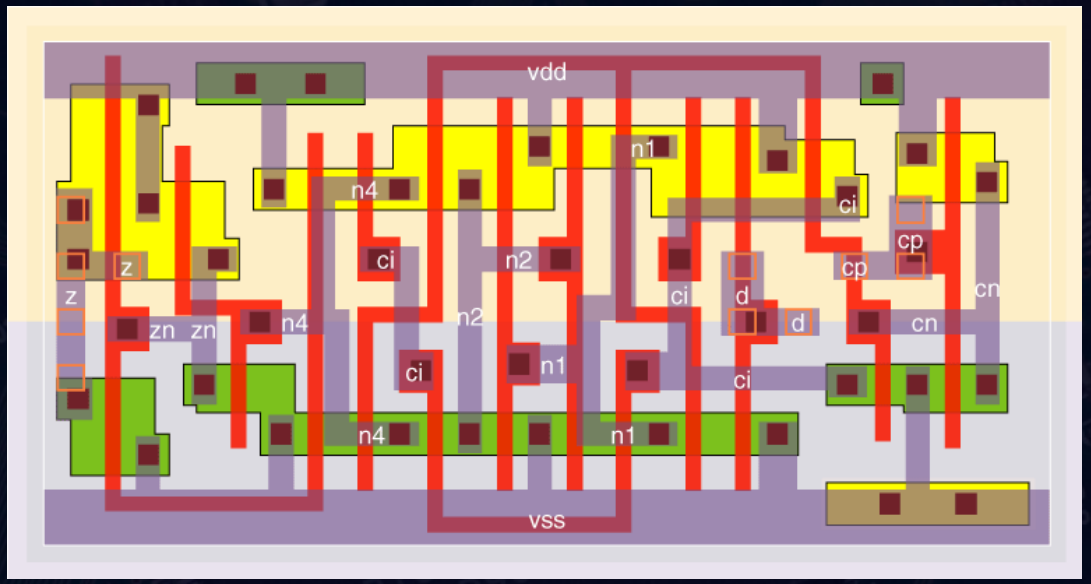
FPGA

Fluxo de projeto ASIC *standard cell*

- Análogo ao fluxo com FPGA, porém
  - "Tech-Map" mapeia em uma biblioteca de células



AND 2 entradas



Flip-flop D http://vlsitechnology.org

  - "Place" tem liberdade para encaixar células em qualquer local de uma *grid* distribuída pelo silício
  - "Route" pode passar trilhas de metal onde quer que exista espaço (não limitado a interconexões predefinidas)

Linguagem Bluespec

- Um nível de abstração acima do RTL
- Totalmente sintetizável
  - Ao contrário de Verilog e VHDL (!)
- Suporte completo a parametrização e elaboração estática (inclusive recursão)
- Interfaces de módulos padronizadas e formalizadas no próprio código
- Transações atômicas para evitar conflitos no acesso a interfaces de módulos

Exemplo simples: Algoritmo de Euclid

Calcule o máximo divisor comum entre 15 e 6

y	x	
15	6	
9	6	subtrair
3	6	subtrair
6	3	trocar
3	3	subtrair
0	3	subtrair

resposta

Implementação em Bluespec

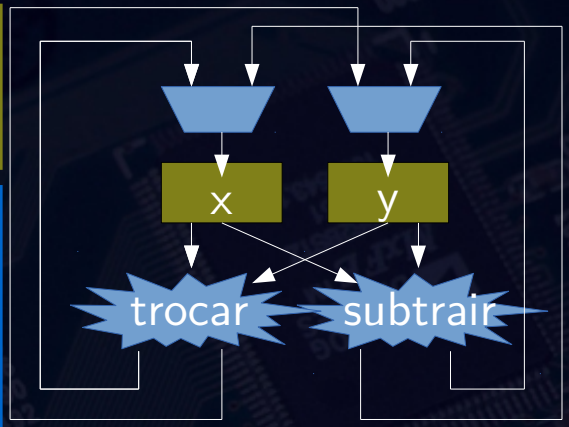
```
module mkEuclid(IEuclid);
  Reg#(Int#(32)) x <- mkRegU;
  Reg#(Int#(32)) y <- mkReg(0);
```
Estado

```
  rule trocar((x > y) && (y != 0));
      x <= y; y <= x;
  endrule
  rule subtrair((x <= y) && (y != 0));
      y <= y - x;
  endrule
```
Comportamento interno

```
  method Action start(Int#(32) a, Int#(32) b) if(y == 0);
      x <= a; y <= b;
  endmethod
  method Int#(32) result() if(y == 0);
      return x;
  endmethod
```
Interface externa

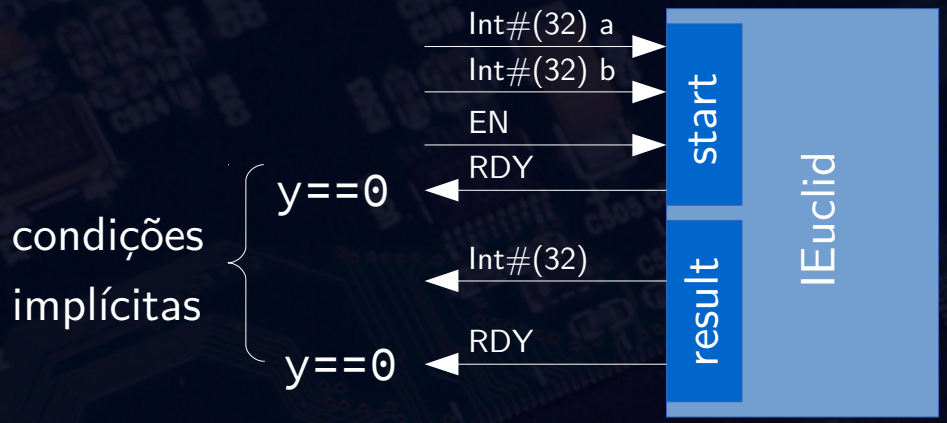
```
endmodule
```

(a==0) ? 0 : b



Algum problema quando a==0?

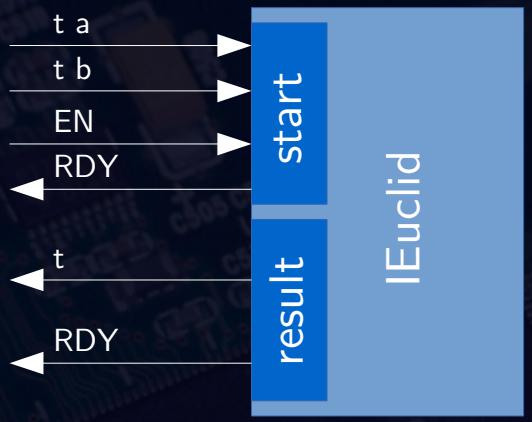
Interface do módulo mkEuclid




```
interface IEuclid;
    method Action start(Int#(32) a, Int#(32) b);
    method Int#(32) result();
endinterface
```

- O módulo pode ser facilmente tornado polimórfico

Interface do módulo mkEuclid




```
interface IEuclid#(type t);
    method Action start(t a, t b);
    method t result();
endinterface
```

- O módulo pode ser facilmente tornado polimórfico ✓
- Muitas implementações diferentes podem fornecer a mesma interface

Outra implementação do módulo

```
module mkEuclidFast(IEuclid#(Int#(32)));
    Reg#(Int#(32)) x <- mkRegU;
    Reg#(Int#(32)) y <- mkReg(0);
    rule trocar_subtrair((x > y) && (y != 0));
        x <= y; y <= x - y;
    endrule
    rule subtrair((x <= y) && (y != 0));
        y <= y - x;
    endrule
    method Action start(Int#(32) a, Int#(32) b) if(y == 0);
        x <= a; y <= (a==0) ? 0 : b;
    endmethod
    method Int#(32) result() if(y == 0);
        return x;
    endmethod
endmodule
```

Usando a interface polimórfica

Combinando as regras trocar e subtrair

Calcula mais rápido? Sim, 31% menos ciclos

Gasta mais recursos? Sim, 40% mais área

Verilog RTL gerado (1)

```
module mkEuclidFast(CLK,
            RST_N,

            start_a,
            start_b,
            EN_start,
            RDY_start,

            result,
            RDY_result);
  input  CLK;
  input  RST_N;

  // action method start
  input  [31 : 0] start_a;
  input  [31 : 0] start_b;
  input  EN_start;
  output RDY_start;

  // value method result
  output [31 : 0] result;
  output RDY_result;

  // signals for module outputs
  wire [31 : 0] result;
  wire RDY_result, RDY_start;

  // register x
  reg [31 : 0] x;
  wire [31 : 0] x$D_IN;
  wire x$EN;

  // register y
  reg [31 : 0] y;
  reg [31 : 0] y$D_IN;
  wire y$EN;

  // rule scheduling signals
  wire WILL_FIRE_RL_subtrair, WILL_FIRE_RL_trocar_subtrair;

  // inputs to muxes for submodule ports
  wire [31 : 0] MUX_y$write_1__VAL_1,
       MUX_y$write_1__VAL_2,
       MUX_y$write_1__VAL_3;

  // remaining internal signals
  wire x_SLE_y___d3;

  // action method start
  assign RDY_start = y == 32'd0 ;

  // rule RL_trocar_subtrair
  assign WILL_FIRE_RL_trocar_subtrair = !x_SLE_y___d3 && y != 32'd0 ;

  // rule RL_subtrair
  assign WILL_FIRE_RL_subtrair = x_SLE_y___d3 && y != 32'd0 ;

  // inputs to muxes for submodule ports
  assign MUX_y$write_1__VAL_1 = x - y ;
  assign MUX_y$write_1__VAL_2 = y - x ;
  assign MUX_y$write_1__VAL_3 = (start_a == 32'd0) ? start_a : start_b ;

  // register x
  assign x$D_IN = EN_start ? start_a : y ;
  assign x$EN = EN_start || WILL_FIRE_RL_trocar_subtrair ;
```

Verilog RTL gerado (2)

```
  // register y
  always@(WILL_FIRE_RL_trocar_subtrair or
      MUX_y$write_1__VAL_1 or
      WILL_FIRE_RL_subtrair or
      MUX_y$write_1__VAL_2 or EN_start or MUX_y$write_1__VAL_3)
  begin
    case (1'b1) // synopsys parallel_case
      WILL_FIRE_RL_trocar_subtrair: y$D_IN = MUX_y$write_1__VAL_1;
      WILL_FIRE_RL_subtrair: y$D_IN = MUX_y$write_1__VAL_2;
      EN_start: y$D_IN = MUX_y$write_1__VAL_3;
      default: y$D_IN = 32'hAAAAAAAA /* unspecified value */ ;
    endcase
  end
  assign y$EN =
         WILL_FIRE_RL_trocar_subtrair || WILL_FIRE_RL_subtrair ||
         EN_start ;

  // remaining internal signals
  assign x_SLE_y___d3 = (x ^ 32'h80000000) <= (y ^ 32'h80000000) ;

  // handling of inlined registers

  always@(posedge CLK)
  begin
    if (!RST_N)
      begin
        y <= `BSV_ASSIGNMENT_DELAY 32'd0;
      end
    else
      begin
        if (y$EN) y <= `BSV_ASSIGNMENT_DELAY y$D_IN;
      end
    if (x$EN) x <= `BSV_ASSIGNMENT_DELAY x$D_IN;
  end
endmodule
```

O que o Bluespec gera automaticamente?

- Sempre lógica combinacional
  - Não afeta especificação ciclo-a-ciclo do hardware

```
rule decr ( x <= y && y != 0 );
   y <= y - x;
endrule : decr

rule swap (x > y && y != 0);
   x <= y; y <= x;
endrule: swap
```

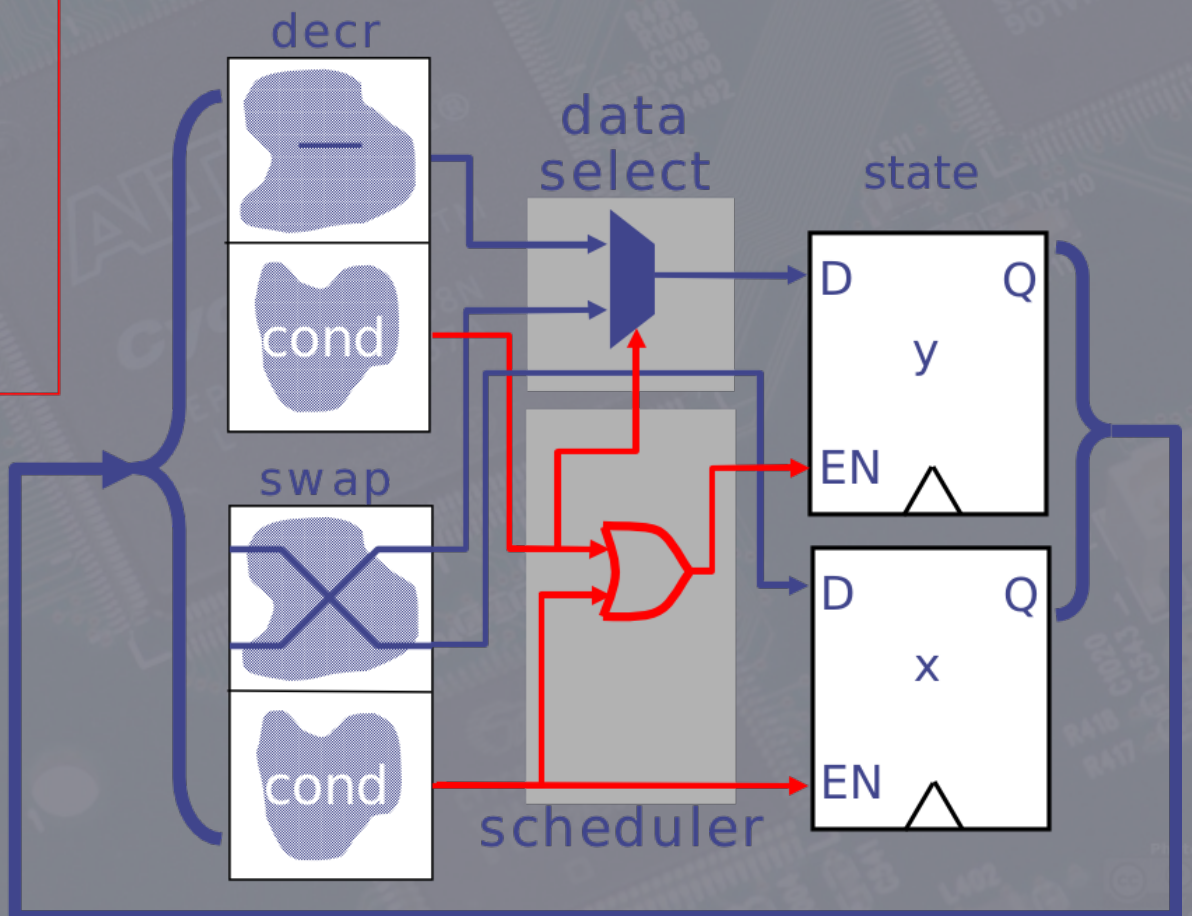


Imagem: Documentação Bluespec – http://bluespec.com/forum/viewtopic.php?t=7

Escrevendo um teste simples

```
import Euclid::*;  // importa tudo do arquivo Euclid.bsv
module mkTest();
    Reg#(Int#(32)) state <- mkReg(0);
    IEuclid#(Int#(32)) euclid <- mkEuclidFast();
    rule comecar(state == 0);
        euclid.start(423, 142);
        state <= 1;
    endrule
    rule finalizar(state == 1);
        $display("Resultado: %d", euclid.result());
        $finish();
    endrule
endmodule
```

Por que precisamos guardar o valor de "state"?

Não seria necessária alguma temporização antes de pegar o resultado?

Teste mais elaborado

```
import Euclid::*;

module mkTest();

    Reg#(Int#(32)) state <- mkReg(0);

    Reg#(Int#(4))  c1    <- mkReg(0);

    Reg#(Int#(7))  c2    <- mkReg(0);

    IEuclid#(Int#(32)) euclid <- mkEuclidFast();

    rule requisitar(state == 0);

        euclid.start(signExtend(c1), signExtend(c2));

        state <= 1;

    endrule

    rule receber(state == 1);

        $display("mmc(%d, %d): %d", c1, c2, euclid.result());

        if(c1 == 7) begin c1 <= 0; c2 <= c2+1; end

                    else c1 <= c1+1;

        if(c1 == 7 && c2 == 63) $finish(); else state <= 0;

    endrule

endmodule
```

Regras a respeito das regras

- Uma regra dispara no máximo uma vez por ciclo
- Uma regra não dispara caso sua condição explícita não seja satisfeita

condição explícita

```
rule finalizar(state == 1);
    $display("Resultado: %d", euclid.result());
    $finish();
endrule
```

Condições implícitas

- Uma regra não dispara caso suas condições implícitas não sejam satisfeitas

```
rule finalizar(state == 1);
    $display("Resultado: %d", euclid.result());
    $finish();
endrule
```

condição implícita de "finalizar"
devido ao uso do método "result"

```
method Int#(32) result() if(y == 0);
    return x;
endmethod
```

Conflito de regras

- Uma regra não dispara se conflitar com outra regra definida como mais urgente.

```
(* descending_urgency = “rule1, rule2” *)
rule regra1(cond1);
    x <= x + 1;
endrule
rule regra2(cond2);
    x <= x - 1;
endrule
```

Se cond1 && cond2, “regra1” dispara, mas “regra2” não dispara, pois a “regra1” é mais urgente.

- Caso duas regras conflitem, e não esteja definida prioridade para as mesmas, o compilador emite um *warning*.

Semântica da composição de regras

- Ao escrever o código, sempre pense em uma regra disparando depois da outra. Não é necessário pensar nas regras disparando ao mesmo tempo.
- Quando existem muitas regras, isso facilita o projeto, pois só é necessário raciocinar em uma regra por vez.
- O compilador agenda o maior número de regras possível para disparar ao mesmo tempo, mas **garantindo** que a semântica acima seja mantida (atomicidade).

Escalonamento de regras

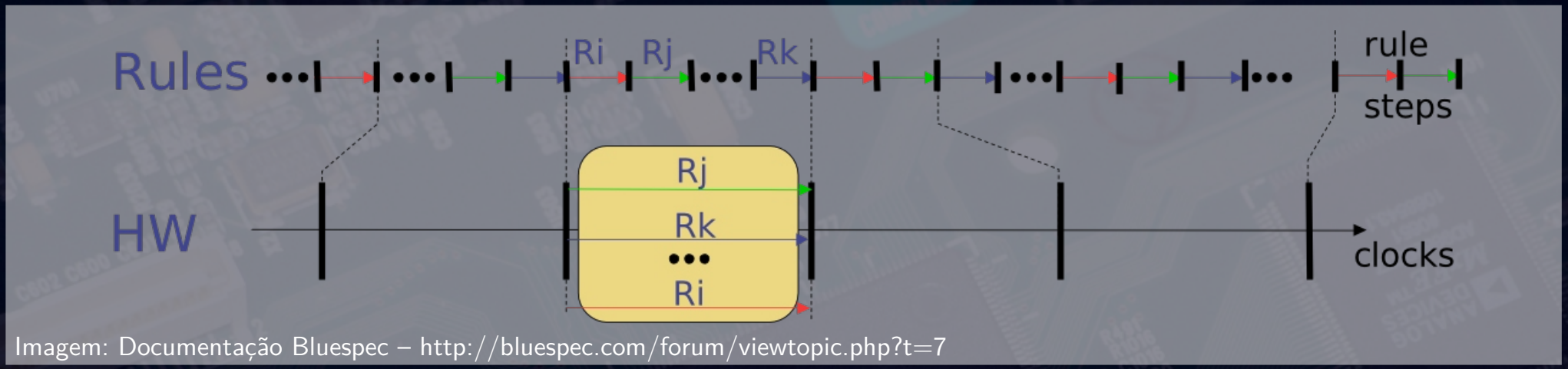


Imagem: Documentação Bluespec – http://bluespec.com/forum/viewtopic.php?t=7

- Regras nunca se comunicam entre si dentro de um mesmo ciclo (a menos que usados módulos de *bypass*, como o Wire).
- O compilador só permite que Ri,Rj,...,Rk executem concorrentemente quando consegue provar que a mudança total de estado é equivalente a Ri<Rj<...<Rk.

Síntese de regras concorrentes

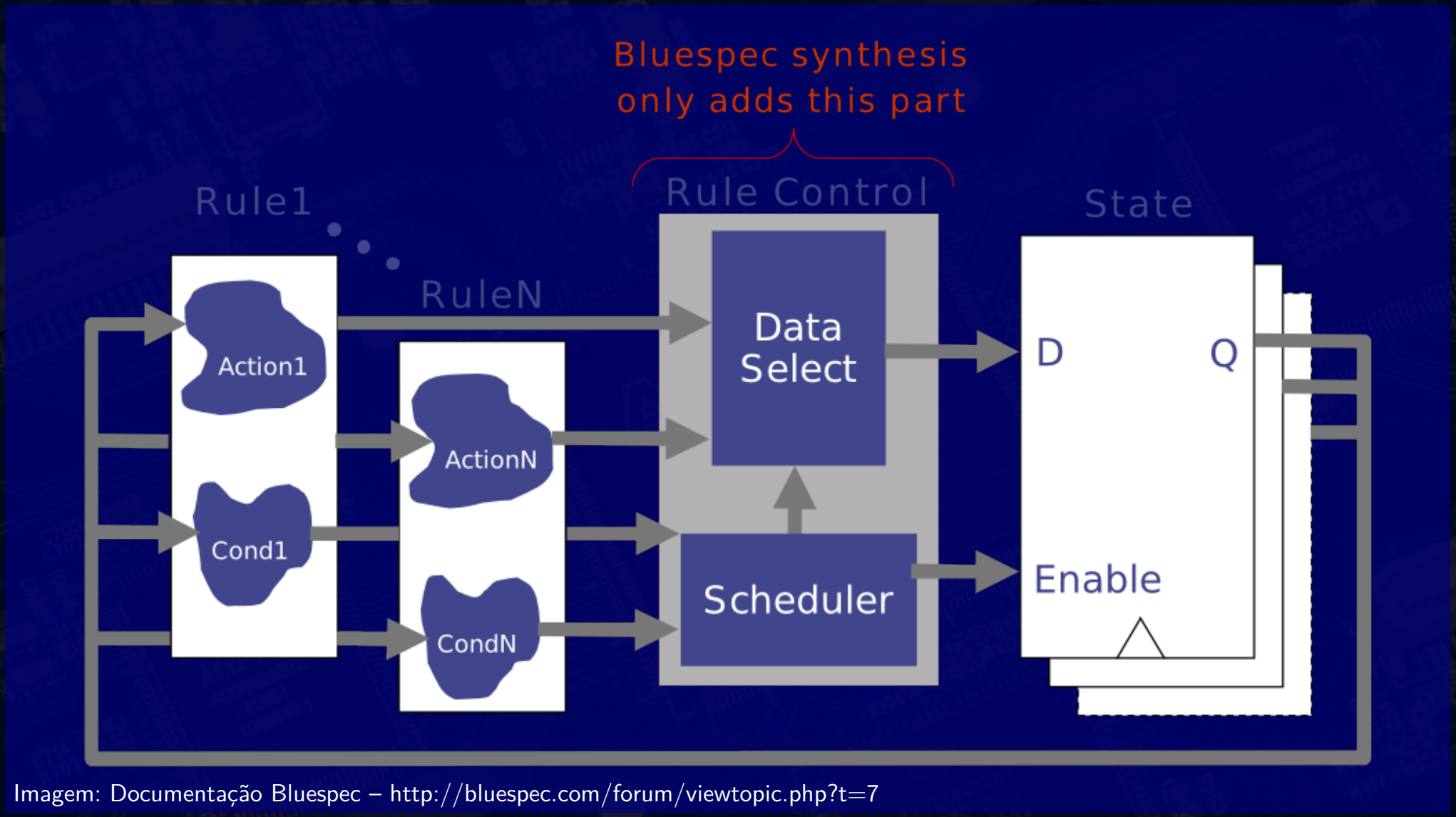


Imagem: Documentação Bluespec – http://bluespec.com/forum/viewtopic.php?t=7

Métodos com efeitos colaterais

- Métodos que alterem o estado do circuito (*side-effects*) devem ser do tipo **Action**.

```
method Action start(Int#(32) a, Int#(32) b) if(y == 0);
    x <= a; y <= (a==0) ? 0 : b;
endmethod
```

- Existe um tipo **ActionValue#(*outrotipo*)**, para encapsular um valor de retorno dentro de uma ação.

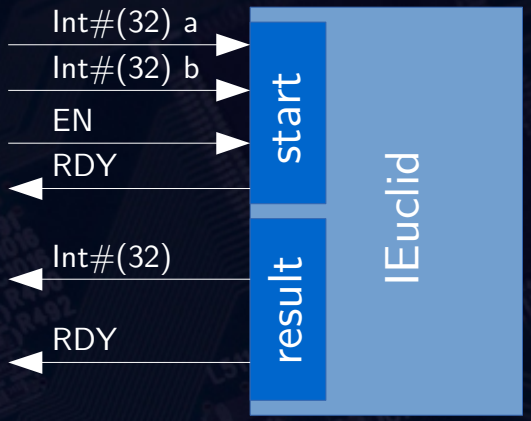
```
method ActionValue#(Bit#(32)) get();
    fifo.deq; return fifo.first;
endmethod
// […]
let valor <- modulo.get;   // desencapsulando o valor
```

Métodos são interfaces padronizadas...

... e não recursos de orientação a objetos!

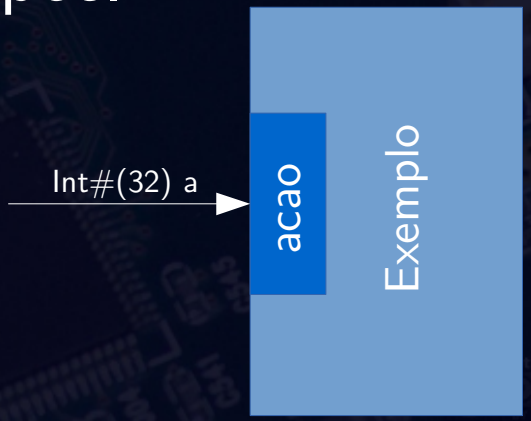
- Pense em **hardware**, não em software.

```
interface IEuclid;
    method Action start(Int#(32) a, Int#(32) b);
    method Int#(32) result();
endinterface
```



- É possível remover os sinais de controle caso desejado. Útil para integrar com circuitos não-Bluespec.

```
interface Exemplo;
    (* always_ready, always_enabled *)
    method Action acao(Int#(32) a);
endinterface
```



Outro Exemplo: RC4

- Gerador criptográfico de números pseudoaleatórios
- Estado: vetor **S[.]** de 256 bytes, ponteiros **i** e **j**

Pseudocódigo: inicialização a partir de uma chave "key"

```
for i from 0 to 255
    S[i] := i
endfor
j := 0
for i from 0 to 255
    j := (j + S[i] +
          key[i mod keylength])
         mod 256
    swap(&S[i],&S[j])
endfor
```

Pseudocódigo: geração da saída do algoritmo

```
i := 0
j := 0
while GeneratingOutput:
    i := (i + 1) mod 256
    j := (j + S[i]) mod 256
    swap(&S[i],&S[j])
    output S[(S[i] + S[j]) mod 256]
endwhile
```

http://en.wikipedia.org/w/index.php?title=RC4&oldid=306386334

Primeira tentativa: Register File

- Arquitetura projetada para fornecer um byte de saída a cada dois ciclos.
- Armazenamento do estado **S[.]**: Register File da biblioteca do Bluespec, com 5 portas de leitura e 1 de escrita.
- Primeiro teste de sintetização: clock máximo de apenas 6.9 MHz em uma FPGA Cyclone II.
- Projeto descartado. Maiores testes não foram realizados.

Segunda tentativa: Block RAM

- Arquitetura projetada para fornecer um byte de saída a cada quatro ciclos.
- Armazenamento do estado **S[.]**: Block RAM da biblioteca do Bluespec, com 2 portas.
- Resultado: Sintetizado com clock máximo de 107 MHz em uma FPGA Cyclone II.
- Throughput: $>$ 200Mbit/s

Implementação do RC4 (1)

```
import FIFO::*;
import GetPut::*;
import StmtFSM::*;
import BRAM::*;
import Vector::*;

typedef Bit#(8) Byte;

interface RC4;
    method Action setKey(Bit#(128) k);
    method Action stop;
    interface Get#(Byte) prng;
endinterface

(* synthesize *)
module mkRC4(RC4);
    BRAM2Port#(Byte, Byte) mem <- mkBRAM2Server(defaultValue);
    Reg#(Vector#(16, Byte)) key <- mkRegU;
    FIFO#(Byte) stream <- mkFIFO;

    Reg#(Bool) loopDone <- mkRegU;
    Reg#(Byte) i <- mkRegU;
    Reg#(Byte) j <- mkRegU;
    Reg#(Byte) si <- mkRegU;
    Reg#(Maybe#(Byte)) fixLateWrite <- mkRegU;
```

Implementação do RC4 (2)

```
function makeReq(wr, addr, data) =
    BRAMRequest{write: wr, address: addr, datain: data,
                responseOnWrite: False};
function wrReq = makeReq(True);
function rdReq(addr) = makeReq(False, addr, ?);
```

```
Stmt stmt = seq
    //--------------------------
    // Key Scheduling Algorithm
    //--------------------------

    action i <= 0; loopDone <= False; endaction

    while(!loopDone)
      action
        mem.portA.request.put(wrReq(i   , i   ));
        mem.portB.request.put(wrReq(i+1, i+1));
        if(i == 254) loopDone <= True;
        i <= i + 2;
      endaction
```

Implementação do RC4 (3)

```
action i <= 0; j <= 0; loopDone <= False; endaction

while(!loopDone)
    seq
      mem.portA.request.put(rdReq(i));
      action
        let siNew <- mem.portA.response.get();
        Byte jNew = j + siNew + key[i & 15];
        mem.portA.request.put(rdReq(jNew));
        si <= siNew;
        j  <= jNew;
      endaction
      action
        let sj <- mem.portA.response.get();
        if(i != j)   // only swap if source != destination
          begin
            mem.portA.request.put(wrReq(i, sj));
            mem.portB.request.put(wrReq(j, si));
          end
        if(i == 255) loopDone <= True;
        i <= i + 1;
      endaction
    endseq
```

Implementação do RC4 (4)

```
//-----------------------------------
// Pseudo Random Generation Algorithm
//-----------------------------------
action i <= 0; j <= 0;
       loopDone <= False; endaction

while(!loopDone)
    seq
      action
        mem.portA.request.put(rdReq(i+1));
        i <= i+1;
      endaction
      action
        let siNew <- mem.portA.response.get();
        Byte jNew = j + siNew;
        mem.portA.request.put(rdReq(jNew));
        si <= siNew;
        j  <= jNew;
      endaction
      action
        let sjNew <- mem.portA.response.get();
        Byte addr = si + sjNew;
        if(addr == j)
            fixLateWrite <= tagged Valid si;
        else if(addr == i)
            fixLateWrite <= tagged Valid sjNew;
        else
            fixLateWrite <= tagged Invalid;
        mem.portA.request.put(rdReq(addr));
        mem.portB.request.put(wrReq(i, sjNew));
      endaction
      action
        let prn <- mem.portA.response.get();
        mem.portA.request.put(wrReq(j, si));
        stream.enq(fromMaybe(prn, fixLateWrite));
      endaction
    endseq
```

Implementação do RC4 (5)

```
        stream.clear;

    endseq;

    FSM fsm <- mkFSM(stmt);

    method Action stop;
        loopDone <= True;
    endmethod

    method Action setKey(Bit#(128) k);
        key <= unpack(k);
        fsm.start;
    endmethod

    interface Get prng = fifoToGet(stream);
endmodule
```

Módulo de testes para o RC4

```
import StmtFSM::*; import RC4::*; import GetPut::*;

(* synthesize *)
module mkTestRC4();
    RC4 rc4 <- mkRC4;
    Reg#(Byte) i <- mkReg(0);

    function genDisplay(Byte num) = seq
        for(i <= 0; i < num; i <= i + 1)
          action
            let x <- rc4.prng.get;
            $display(x);
          endaction
    endseq;

    Stmt stmt = seq
            rc4.setKey(128'he840aa151ccb6dbe26f9c8ea3edc993e);
            GenDisplay(4); rc4.stop;

            $display("---");
            rc4.setKey(128'hdeadbeefdeadbeefdeadbeefdeadbeef);
            genDisplay(4); rc4.stop; $display("---");

            rc4.setKey(128'he840aa151ccb6dbe26f9c8ea3edc993e);
            genDisplay(8);
        endseq;
    mkAutoFSM(stmt);
endmodule
```

Onde aprender mais?

- Materiais nos quais estes slides foram baseados
  - http://bluespec.com/forum/viewtopic.php?t=7
  - http://csg.csail.mit.edu/6.375
- Outros materiais
  - http://wiki.bluespec.com/Home/BSV-Documentation
- Projetos (802.11a, H.264, MD6, ReedSolomon)
  - http://www.opencores.org