Logic

EE599-001 & EE699-010, Spring 2026

Hank Dietz

`http://aggregate.org/hankd/`

Quantum is Programmed at the (Qu)Bit/Gate Level

- Gate-level optimization is critical
  - Very few qubits
  - Very limited gate sequence depth
  - Additional constraints on gates
- Quantum gates
  - Different; optimization is harder
  - Some quantum gates don't have conventional analogs (e.g., √NOT); we'll start with the ones that do

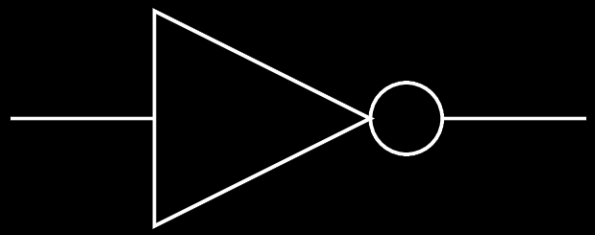
Digital Logic & Optimization

- Conventional gates
  - Basic gate types, fanout
  - Logic optimization
- Adiabatic/Reversible logic
  - Gate types and mechanisms
  - No fanout; ancilla
  - Logic optimization
- A little Verilog project...

Conventional Gates: NOT

- `NOT(A)`, `!A`, `~A`

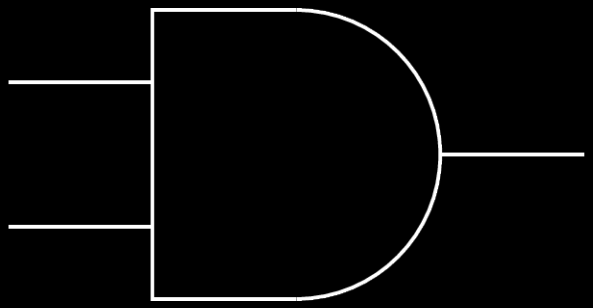
A	Q
0	1
1	0



Conventional Gates: AND

- `AND(A,B)`, `A&B`, `A*B`, `AB`

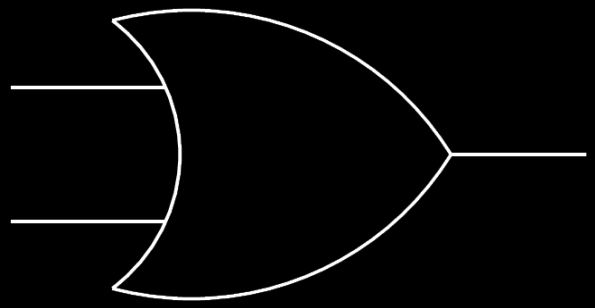
```
A B | Q
0 0 | 0
0 1 | 0
1 0 | 0
1 1 | 1
```



Conventional Gates: OR

- `OR(A,B)`, `A|B`, `A+B`

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1



Logic Optimization

- Sum of Products form
  - OR over ANDs, with NOTs
  - Rule-based simplifier here:
    `https://www.boolean-algebra.com/`
  - Karnaugh maps
  - Quine-McClusky:
    `https://www.mathematik.uni-marburg.de/~thormae/lectures/ti1/code/qmc/`
  - Espresso (in Python EDA):
    `https://pyeda.readthedocs.io/en/latest/2llm.html`
- Don't actually optimize for 2-input gates...

We'll Be Using Verilog...

- IEEE 1364-2005
- My slides about Verilog
  `https://aggregate.org/CPE380/slidesS26verilog.pdf`
- Various Verilog materials online, e.g.:
  `http://www.asic-world.com/verilog/`
- Icarus Verilog
  `https://en.wikipedia.org/wiki/Icarus_Verilog`
- My CGI interface to Icarus Verilog
  `https://aggregate.org/cgi-bin/iver.cgi`

We'll Be Using Verilog...

- IEEE 1364-2005
- **Verilog HDL** by Samir Palnitkar, ISBN-978-0132599702
- Various Verilog materials online, e.g.:
  `http://www.asic-world.com/verilog/`
- Icarus Verilog
  `https://en.wikipedia.org/wiki/Icarus_Verilog`
- My CGI interface to Icarus Verilog
  `https://aggregate.org/cgi-bin/iver.cgi`

We'll Be Using Verilog...

- We will use Verilog to implement a gate-level circuit design & test it
- Test will be exhaustive, using an oracle
- Let's run through what that looks like with a simple example from CPE380:
  a **ripple carry adder**
- First fixed size, then parametric...

A 1-Bit Half Adder (HA)

```
// Half Adder
module ha(sum, cout, a, b);
output sum, cout;
input a, b;
assign sum = a ^ b;     // sum
assign cout = a & b;    // cout
endmodule
```

A 1-Bit Half Adder (HA)

```
// Half Adder
module ha(sum, cout, a, b);
output sum, cout;
input a, b;
xor(sum, a, b);         // sum
and(cout, a, b);        // cout
endmodule
```

A 1-Bit Full Adder (FA)

```
// Full Adder
module fa(sum, cout, a, b, cin);
output sum, cout;
input a, b, cin;
wire aorb, gen, prop;
xor(sum, a, b, cin);    // sum
and(gen, a, b);         // generate
or(aorb, a, b);         // propagate
and(prop, aorb, cin);
or(cout, gen, prop);    // cout
endmodule
```

A 1-Bit Full Adder (FA)

```
`define CSWAP(C,A,B) {A,B}=(C?{B,A}:{A,B})

// Full Adder
module cswapfa(sum, cout, ain, bin, cin);
input ain, bin, cin;
output reg sum, cout; // total 5 registers
reg a, b, g;
always @ (*) begin
  sum = 0; cout = cin; a = ain; b = bin; g = 1;
  `CSWAP(a, sum, g);
  `CSWAP(b, sum, g);
  `CSWAP(cout, sum, g);
  `CSWAP(sum, cout, g);
  `CSWAP(b, cout, g);
end
endmodule
```

A 1-Bit Full Adder (FA)

```
`define CSWAP(C,A,B) {A,B}=(C?{B,A}:{A,B})

// CSWAP Full Adder
module cswapfa(sum, cout, a, b, cin);
input ain, bin, cin;
output reg sum, cout; // total 3 writable registers
reg g;
always @ (*) begin
  sum = 0; cout = cin; g = 1;
  `CSWAP(a, sum, g);
  `CSWAP(b, sum, g);
  `CSWAP(cout, sum, g);
  `CSWAP(sum, cout, g);
  `CSWAP(b, cout, g);
end
endmodule
```

An 8-Bit Ripple Carry Adder

```
// Ripple carry addition, 8-bit
module add8(sum, cout, a, b, cin);
output [7:0] sum;
output cout;
input [7:0] a, b;
input cin;
wire [6:0] lcout;
fa fa0(sum[0], lcout[0], a[0], b[0], cin),
   fa1(sum[1], lcout[1], a[1], b[1], lcout[0]),
   fa2(sum[2], lcout[2], a[2], b[2], lcout[1]),
   fa3(sum[3], lcout[3], a[3], b[3], lcout[2]),
   fa4(sum[4], lcout[4], a[4], b[4], lcout[3]),
   fa5(sum[5], lcout[5], a[5], b[5], lcout[4]),
   fa6(sum[6], lcout[6], a[6], b[6], lcout[5]),
   fa7(sum[7], cout, a[7], b[7], lcout[6]);
endmodule
```

Let's Test It: `testbench`

```
module testbench;
reg [7:0] a, b, refsum; wire [7:0] sum;
reg cin, refcout; wire cout; integer tested=0, wrong=0;

add8 uut(sum, cout, a, b, cin);       // unit under test

initial begin a=0; repeat (256) begin b=0;
  repeat (256) begin cin=0; repeat (2) #1 begin
    {refcout, refsum} = a + b + cin;  // oracle
    tested=tested+1;
    if ((refcout != cout) || (refsum != sum)) begin
$display("Wrong: %d+%d+%d is {%d,%d}, but got {%d,%d}",
      a, b, cin, refcout, refsum, cout, sum);
    wrong=wrong+1;
  end cin=1; end b = b + 1; end a = a + 1; end
$display("%d cases tested, %d wrong", tested, wrong);
end endmodule
```

Let's Really Test It...

- Can run it here:
  `http://aggregate.org/EE380/aluripple8.html`
- Notice that this is an exhaustive test
  - All 131,072 cases are tried ($2^8 \times 2^8 \times 2$)
  - Exhaustive testing quickly becomes less feasible as number of input bits grows

A Parametric Ripple Adder

```
// Ripple carry addition, BITS-bit
module add(sum, cout, a, b, cin);
parameter BITS=8;
output [BITS-1:0] sum;
output cout;
input [BITS-1:0] a, b;
input cin;
wire [BITS:0] c;  // temporary (local) wires

genvar i;
generate for (i=0; i<BITS; i=i+1) begin:fas
  // full adders named fas[i].myfa
  fa myfa(sum[i], c[i+1], a[i], b[i], c[i]);
end endgenerate

assign c[0] = cin;      // first carry in
assign cout = c[BITS];  // last carry out
endmodule
```

Let's Test That One Too...

- Can run it here:
  `http://aggregate.org/EE380/alurippleBITS.html`
- Notice that the entire Verilog program is **just one line longer** than the 8-bit-only version
- This version essentially **generates the exact same gates**; there's no hardware cost to being parametric

All 16 Functions of 2 Inputs

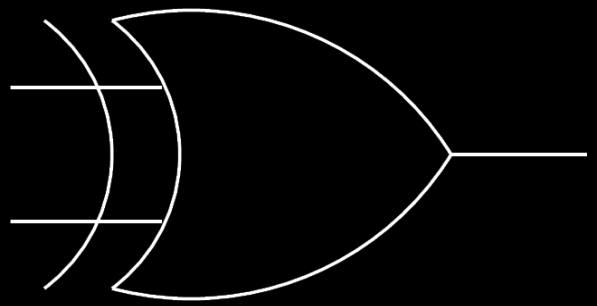
- Enumerating values in order of {**B**,**A**}:

```
0000  0                 1000  !(A|B)
0001  A&B               1001  !(A^B), A==B
0010  A&!B              1010  !B
0011  A                 1011  A|!B, A>=B
0100  B&!A              1100  !A
0101  B                 1101  B|!A, B>=A
0110  A^B, A!=B         1110  !(A&B)
0111  A|B               1111  1
```

Conventional Gates: XOR

- `XOR(A,B)`, `A^B`, `A!=B`, `A+B`, `odd parity`

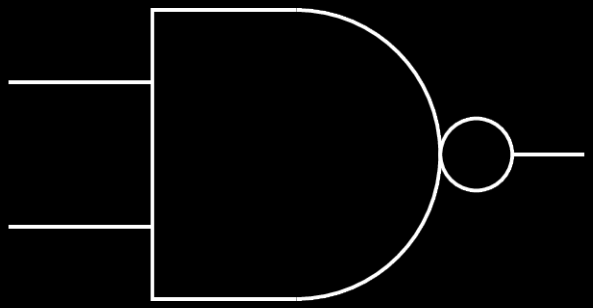
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

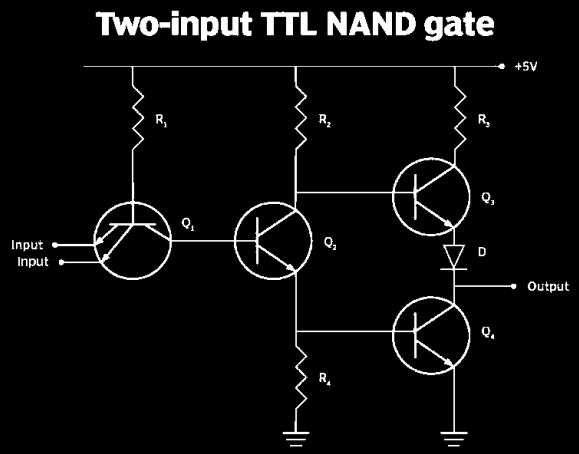


Conventional Gates: NAND

- `NAND(A,B)` is `NOT(AND(A,B))`

A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0



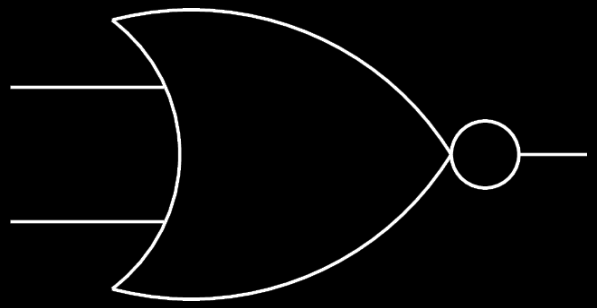


- Note TTL Totem Pole output

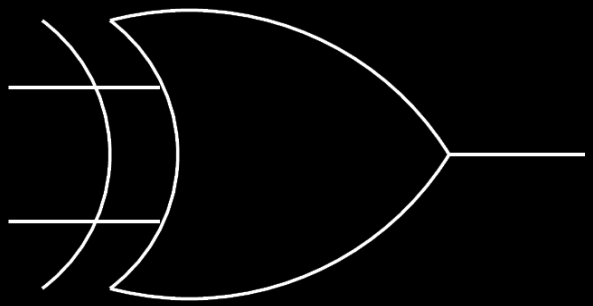
Conventional Gates: NOR

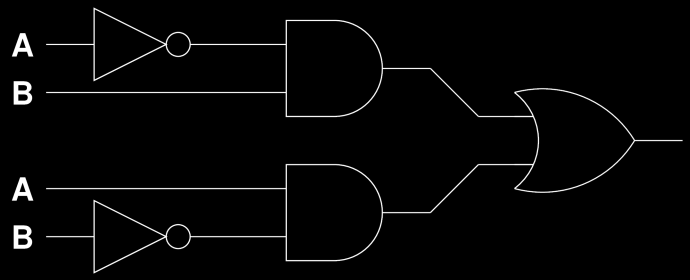
- `NOR(A,B)` is `NOT(OR(A,B))`

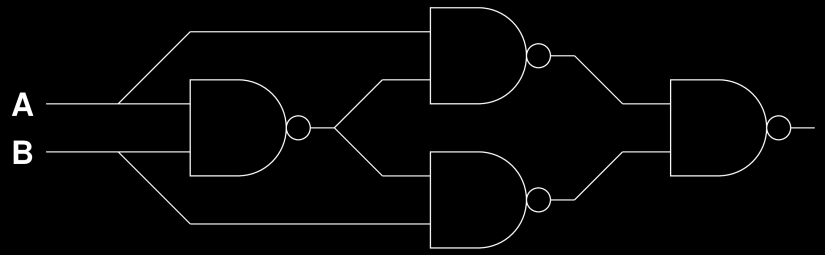
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

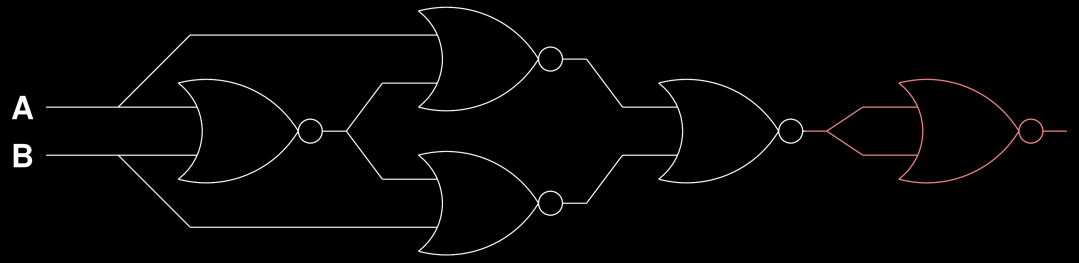


Constructing XOR

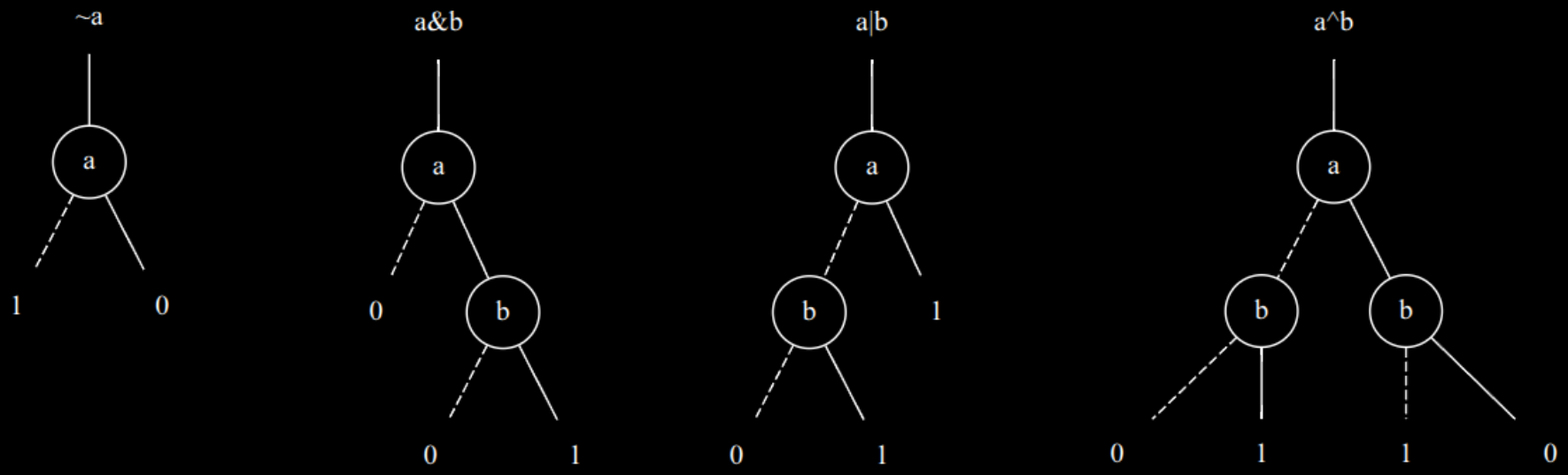








BDD Analysis/Optimization

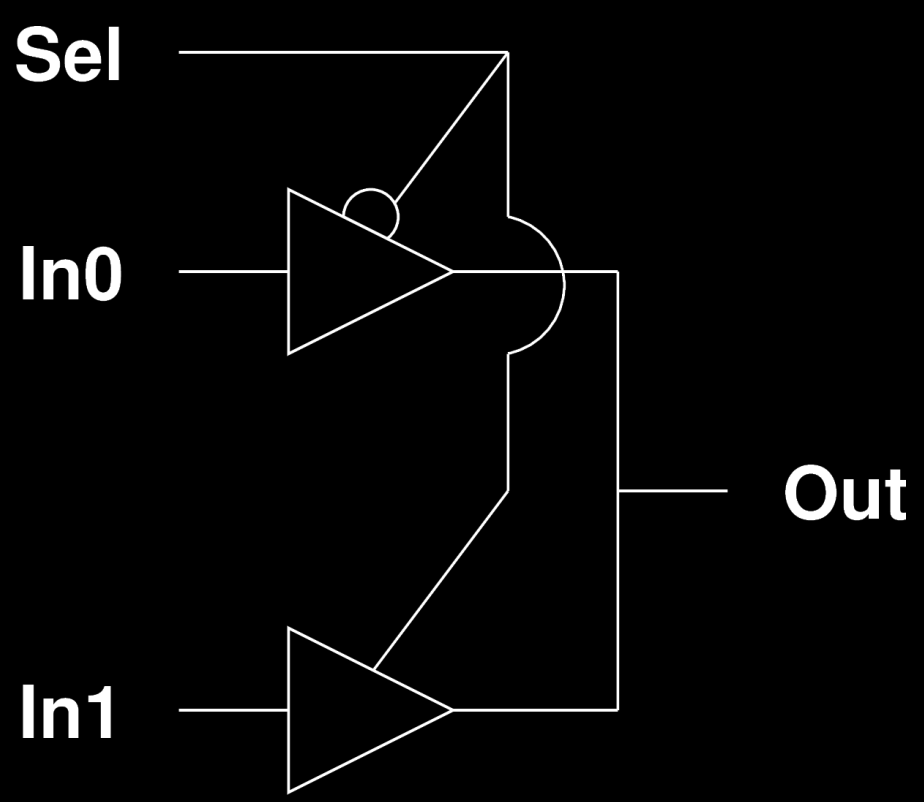


- **BDD** (Binary Decision Diagram) is commonly used for proofs and logic optimization
- **Biddy** is a library for BDD optimization
  `https://aggregate.org/cgi-bin/biddy.cgi`

BDD Gate is ITE (If-Then-Else)

- `ITE(S,B,A)` is `MUX(S,A,B)`,`(S?B:A)`

S	A	B	Q
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



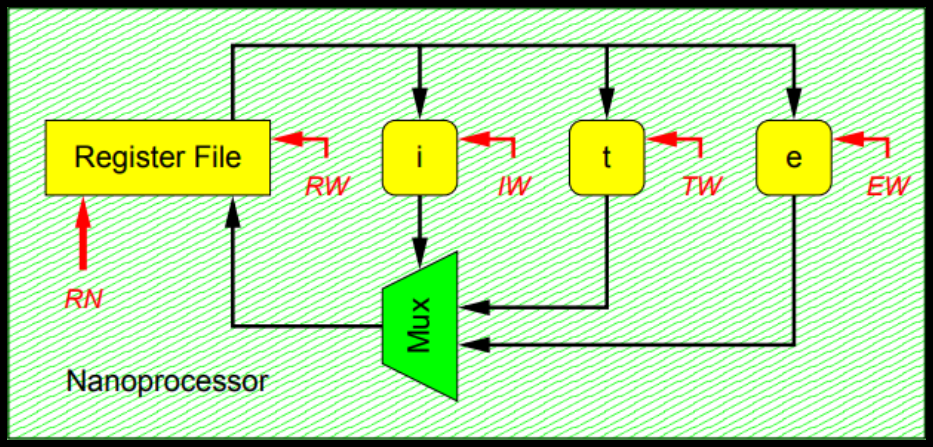
Conventional Gates: ITE

- `ITE(S,B,A)` is a universal gate:

```
NOT(A)          (A?0:1)
AND(A,B)        (A?B:0)
OR(A,B)         (A?1:B)
XOR(A,B)        (A?(B?0:1):B)
```

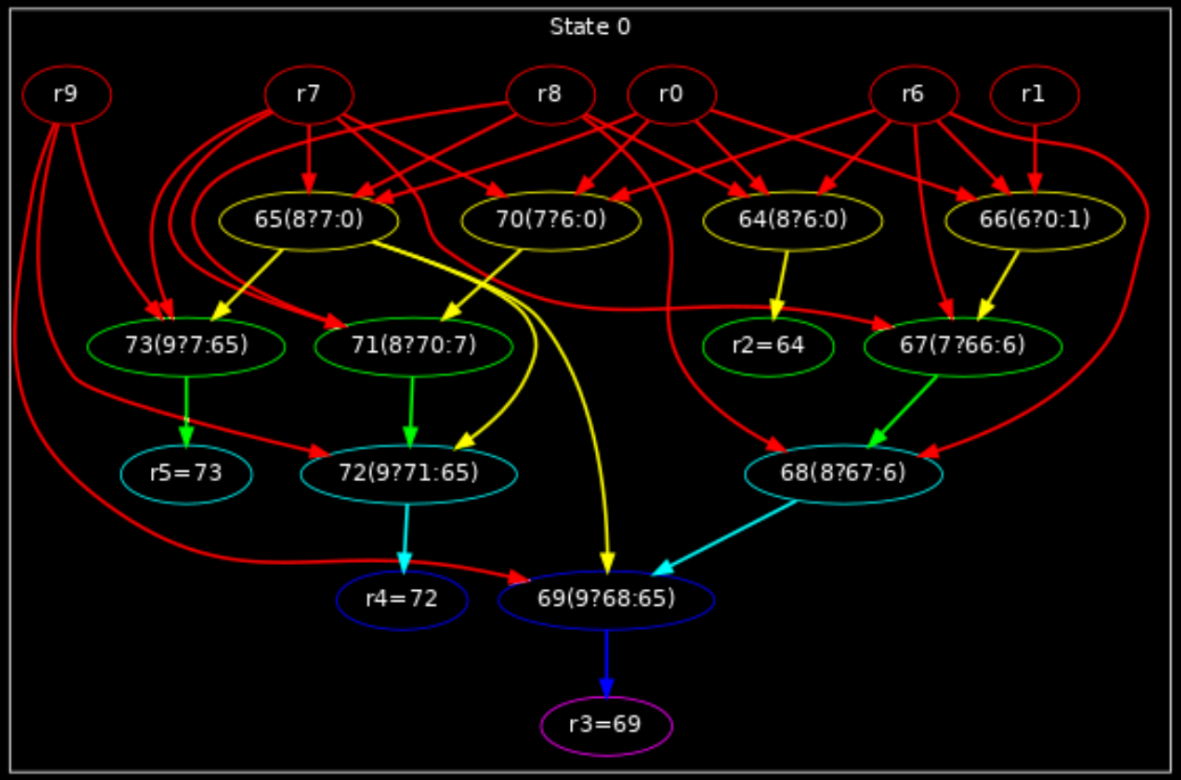
- Also used as BDD (Binary Decision Diagram)

BitC Language & Compiler



- **BitC** is a C dialect created for **KY Architecture Nanocontrollers**: **KITE** (**KY If-Then-Else**)
  `https://aggregate.org/KYARCH/`
- **bitcc** is the BitC compiler generating ITEs
  `https://aggregate.org/cgi-bin/bitcc.cgi`

A BitC Example



- `int:4 a; int:2 b, c; a=b*c;`
- [r0]=0, [r1]=1, [r5:r2]=a, [r7:r6]=b, [r9:r8]=b

  `https://aggregate.org/cgi-bin/bitcc.cgi`

ITEs in Verilog

- Verilog has the `?:` operator
  - We want to *sequence* gates, so we'll use `reg`s and assignments in an `always`
  - Make a `SITE` macro: Store ITE
- Our example multiplies 3-bit numbers
  - `int:3 a,b,c; a=b*c;`

    `https://aggregate.org/cgi-bin/bitcc.cgi`
  - Here's the Verilog code:

    `https://aggregate.org/QC/sitemul.html`

Adiabatic Logic

- Thermodynamically reversible gates
  - Fredkin & Toffoli, 1978: used caps+inductors
  - Hall 1992 & Merkle 1992: 1st reversible comb.
  - Younis & Knight, 1993: 1st reversible FSM
  - Group at MIT, 1995-1999: Pendulum, etc.
  - More in Flordia, Notre Dame, UK, etc.; Spring 2019 CPE480 AXA:
    https://aggregate.org/EE480/axa.html

- Gates can operate forward or reverse…
  can even build instruction set to do that

Reversible Instructions

- Have inverses…
  - XOR (but **NOT** AND, OR!)
  - Rotates (but **NOT** Shifts!)
  - Add, Subtract
  - Exchange (but **NOT** Store or Load!)
  - Jumps that note where they came from (land)
- A^=B is reversible, but A^=A is **NOT**
- Using only such ops, code can execute forward or backward

Energy Recovery

- Power supply is phased (i.e., a clock)
  - Fanout == 1
  - Send energy forward, then recover it:
    think sending +1, then recovering -1
  - Can do this with fairly ordinary gates,
    if you don’t care about running in reverse
  - Awkward to scale up

Symmetric Outputs

- Every signal takes two lines
  - Fanout == 1
  - Output logic 0 is {0,1}, logic 1 is {1,0}
  - A lot of extra wires...
  - Can be applied to conventional gates if you don’t need to run in reverse

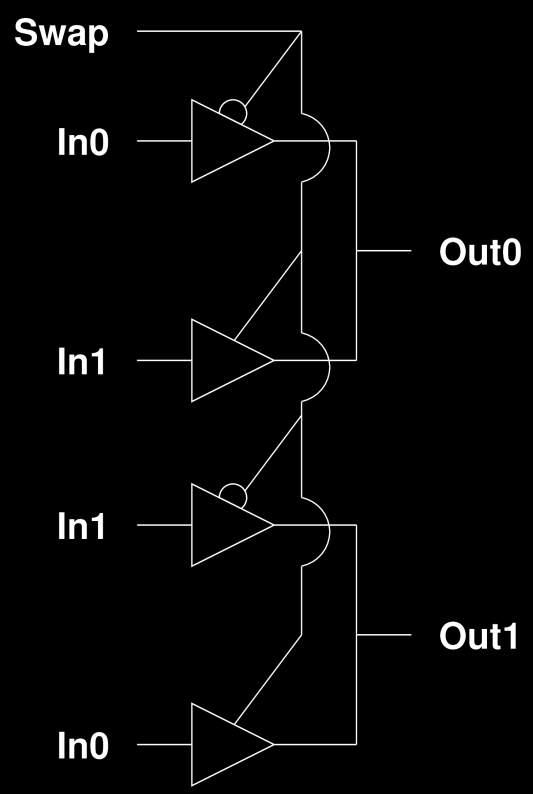
Billiard-Ball Conservancy

- Number of 1's input == 1's output
  - Fanout == 1
  - Requires special gates, e.g., **CSWAP**, Conditional SWAP, aka **Fredkin** (which is also known as a Fredkin gate)
  - Easily reversible execution

CSWAP (Fredkin) Gate

- `CSWAP(S,A,B)` is `{(S?B:A),(S?A:B)}`

S	A	B	QA	QB
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	1
1	0	0	0	0
1	0	1	1	0
1	1	0	0	1
1	1	1	1	1



CSWAPs in Verilog

- Verilog modeling of CSWAP
  - *Sequence* of gate operations
  - Make and use a **`CSWAP`** macro:

```
`define CSWAP(S,A,B) {A,B}=(S?{B,A}:{A,B})
```

  - No assignments outside the macro except initialization, no repeated operands
  - Goal: same function, minimum ancilla with unit fanout (*sequential reuse* is OK)
- Here is a 1-bit Full Adder using CSWAP:

https://aggregate.org/QC/cswapfa.html

How To Copy A Value?

- ```
  copy=0;
  notcopy=1;
  `CSWAP(S,copy,notcopy);
  ```

  Note that this also makes complement…
  but `copy` and `notcopy` are essentially ancilla
  ```

Your Project

- Start with the SITE code:
  `https://aggregate.org/QC/sitemul.html`
- That code uses a big array of bits and creates a new bit for each result… CSWAP doesn't
- Rewrite `module mul` so it uses only CSWAP
  - Use as few CSWAP as possible
  - Use as few ancilla as possible