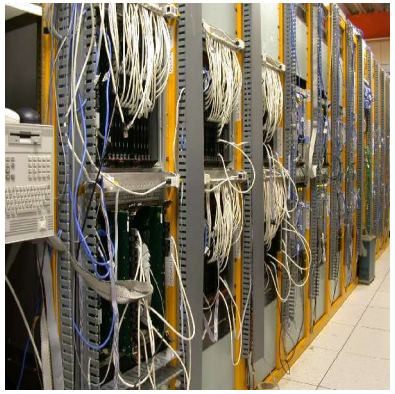
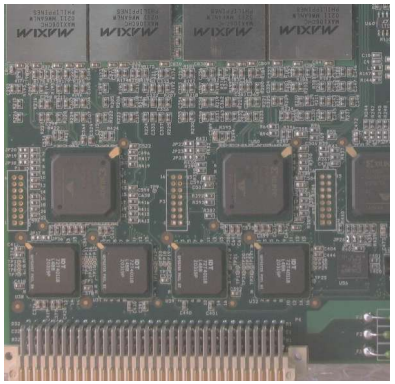
NEPPSR 2005

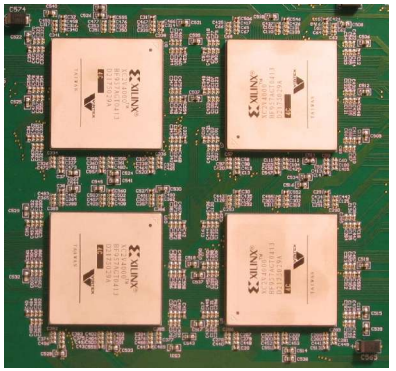
Trigger and DAQ Electronics

Part 2 – Programmable Logic

Eric Hazen, Boston University







Programmable Logic Introduction

For HEP applications we typically use FPGAs (Field-Programmable Gate Arrays)

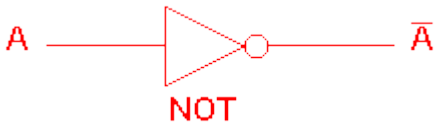
This particular one (XC2V3000) contains:

- 684 inputs and outputs
- 32,256 logic cells
  - any arbitrary function of 4 inputs
  - plus a register (D flip-flop)
- 96 18k bit dual-port RAMs
- 96 18x18 multipliers
- 12 digital clock managers
  - ... and various other goodies

What is all this stuff and how do we use it?

Boolean Logic

Basic Gates



aka INVERTER

Truth Table

NOT gate	
A	$\overline{A}$
0	1
1	0

not A (VHDL)
other notations:
$\overline{A}$ /A !A

2 Input AND gate		
A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

A and B
AB A*B A•B A&B

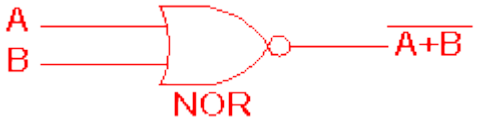
2 Input OR gate		
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

A or B
A+B A|B

Derived Gates

2 Input NAND gate		
A	B	$\overline{A.B}$
0	0	1
0	1	1
1	0	1
1	1	0

A nand B
$\overline{AB}$ (A*B) $\overline{A•B}$!(A&B)



2 Input NOR gate		
A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

A nor B
$\overline{A+B}$ $\overline{A|B}$!(A|B)

2 Input EXOR gate		
A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

A xor B
A⊕B A^B

All digital logic can be described using the operations above.

Boolean Identities

The usual algebraic laws (commutative, associative, distributive, identity) apply:

```
A and B = B and A
A or B = B or A
(A and B) and C = A and (B and C)
    etcetera
```

De Morgan's Theorem:

```
not(A and B) = not(A) or not(B)
not(A or B) = not(A and B)
```

De Morgan's Theorem essentially states that you can exchange AND with OR in an expression provided that all the inputs and outputs are inverted.

Asynchronous Logic

Any logic composed of basic or derived gates without memory or latches is called asynchronous logic. The outputs of a circuit will change to reflect changes in input state (plus a certain *propagation delay*).

Asynchronous logic is also known as combinatorial logic.

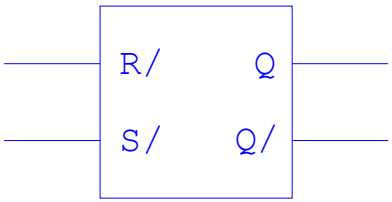
In FPGAs, typically a “look-up table” is used for combinatorial logic. For example, a table which stores all possible functions of 4 bits -----> (You can fill in the 'Y' column with anything)

Traditionally much emphasis in the classroom was placed on simplification or “minimization” of complex boolean expressions. This is largely handled by software now, and is typically not necessary when working with FPGAs.

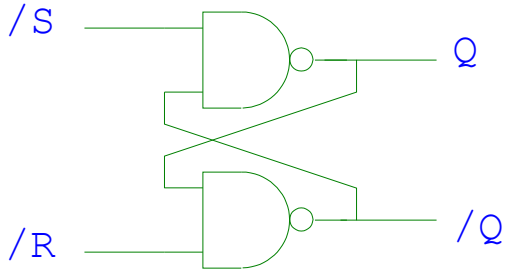
A	B	C	D	Y
0	0	0	0	-
0	0	0	1	-
0	0	1	0	-
0	0	1	1	-
0	1	0	0	-
0	1	0	1	-
0	1	1	0	-
0	1	1	1	-
1	0	0	0	-
1	0	0	1	-
1	0	1	0	-
1	0	1	1	-
1	1	0	0	-
1	1	0	1	-
1	1	1	0	-
1	1	1	1	-

Flip-Flops and Synchronous Logic

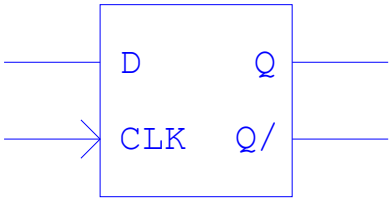
A latch uses positive feedback to make a stable circuit which can store a bit.
The RS Latch is one of the simplest examples:



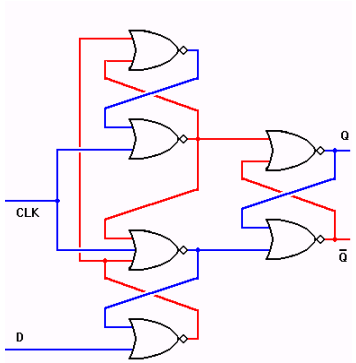
/S	/R	Q	/Q
1	1	Q_0	$/Q_0$
0	1	1	0
1	0	0	1
0	0	1	1



The foundation of synchronous (clocked) logic is the D Flip-Flop.



D	CLK	Q	/Q
-	0	Q_0	$/Q_0$
0	↑	0	1
1	↑	1	0



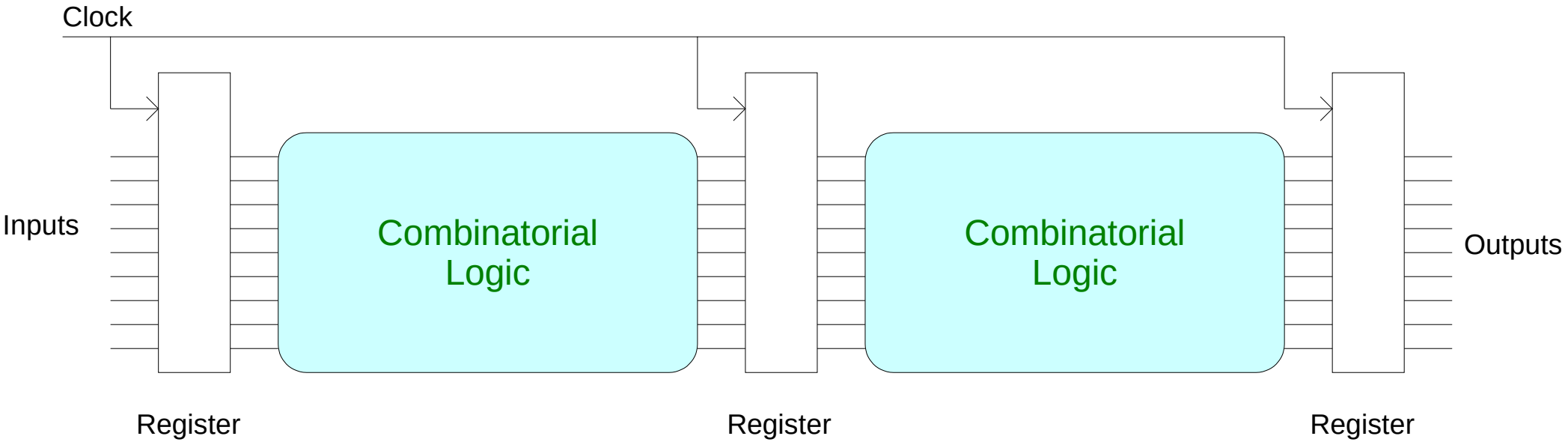
This gate transfers data from D-Q on the *rising edge* (0-1 transition) of the CLK input.
At all other times, Q is unchanging. The D Flip-Flop is a 1-bit memory.

Many D Flop-Flops may be controlled by the same clock to store a number of bits simultaneously. This is called a Register.

Synchronous Logic

Most logic used in trigger/DAQ systems is *pipelined*, and consists of blocks of combinatorial logic between registers.

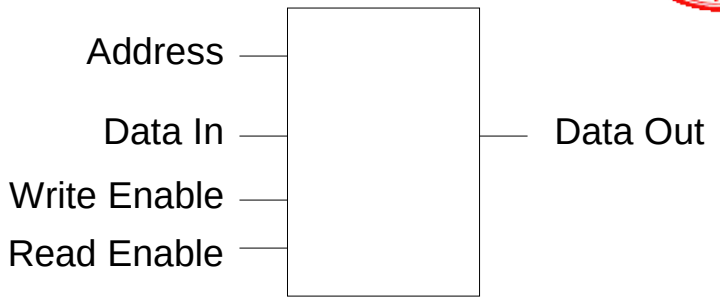
All registers should be clocked by *the same clock*. Often there is an obvious choice, such as the 40MHz RF clock at the LHC, or the 53MHz equivalent at the Tevatron.



A critical requirement is that the *propagation delay* through the combinatorial logic must be less than the time between successive clock edges!

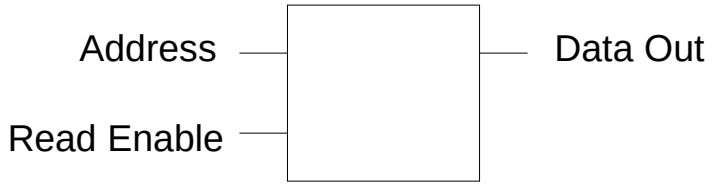
Memories

RAM (Random-Access Memory)
a read/write array of storage cells



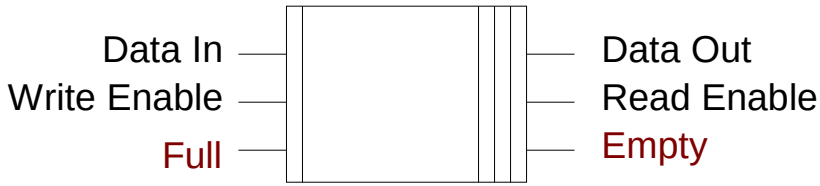
ROM (Read-Only Memory)
an array of fixed values loaded
at system start-up

note: for look-up tables, often the memory is "read mostly" and can be written if required



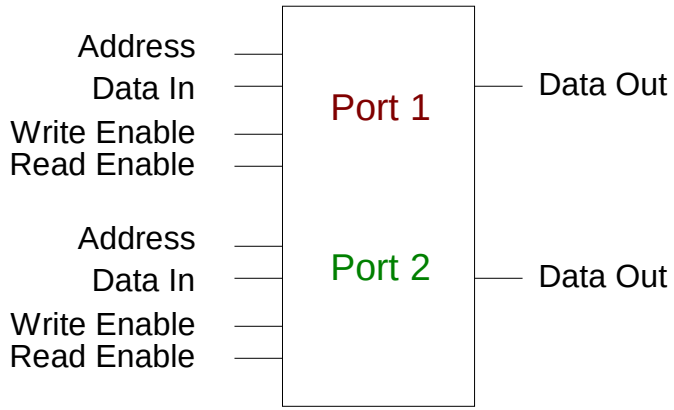
FIFO (First-In First-Out) buffer – an ordered
list in which items are added
at one end and removed from the other

Used commonly where temporary storage is needed in a data stream, or to cross the boundary between parts of the system which use different clocks.



Dual-Port Memories – a type of RAM in which
two different addresses may be
accessed simultaneously.

FPGAs provide these as basic building blocks.
They are used to create all the other types above.



The VHDL Language

Most logic design for Trigger/DAQ systems is done in VHDL (or Verilog, a similar language). These are specialized languages developed to describe logic.

VHDL has many features, but you can get by with just a few. Here we define logic to compare two 2-bit binary values in several ways to illustrate the 4 basic VHDL statement types.

Boolean Equations:

```
aeqb <= (a(0) xor b(0)) nor ((a(1) xor b(1));
```

Simplest and most straightforward way to describe combinatorial logic

Structural (Netlist):

```
u1: xor2 port map(a(0), b(0), x(0));
u2: xor2 port map(a(1), b(1), x(1));
u3: nor2 port map(x(0), x(1), aeqb);
```

Use to “wire together” existing elements. Used particularly for large, pre-defined functions provided by the FPGA vendor.

Concurrent:

```
aeqb <= '1' when a = b else '0';
```

Use for logic with multiple conditions which is difficult to describe in simple Boolean Equations. Functionally, concurrent statements behave as combinatorial logic, that is they evaluate simultaneously. Note that the comparison of multi-bit vectors is handled automatically.

Sequential:

```
aeqb <= '0';
if a = b then aeqb <= '1';
```

Must be used inside a controlling *process*, usually a clocked process which defines synchronous logic. The order of sequential statements is important; the last assigned value to a signal takes precidence.

A Few More VHDL Statements

With...select...when

```
WITH inc SELECT
    outc <= ina WHEN ’0’,
            inb WHEN ’1’,
            inb WHEN OTHERS;
```

Assigment is based on a selection sighal. WHEN clauses must be mutually exclusive.
Always use 'others' clause... in simulation there are values other than '1' and '0'.

Case...When

```
CASE inc IS
    WHEN ’0’ =>    outc <= ina;
    WHEN ’1’ =>    outc <= inb;
    WHEN OTHERS => outc <= ind;
end CASE;
```

Similar to with...select except that any arbitrary assignment may occur after the '=>'.
This is a *sequential* statement, so must appear inside a process.

VHDL Operators and Constants

(subset used for logic synthesis)

Logical (use ieee.std_logic_1164.all)

AND, NAND, OR, NOR, XOR, XNOR, NOT

Relational (use ieee.std_logic_1164.all)

=, /=, <, <=, >, >= Watch out for “double meaning” of <= and =>

Unary arithmetic **-** Arithmetic **+, -** (*, /, mod, rem, ** exist too, but are not synthesizable)

Concatenation – for bit strings **&**

Constants (bit vectors)

```
x"ffe"              (12-bit hexadecimal value)
o"777"              (9-bit octal value)
b"1111_1101_1101"   (12-bit binary value)
```

Constants (integers)

```
16#9fba#            (hexadecimal)
2#1111_1101_1011#   (binary)
```

Don't even think about floating point or character strings!

VHDL Synchronous Logic Example:

Divide-by-10 Counter


```
library IEEE;                           -- standard library declarations
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity divide_by_10 is                  -- declare entity
  port (                                -- (akin to C function declaration)
    clk, rst_n : in  std_logic;         -- system clock, reset (active low)
    div100     : out std_logic);        -- divide-by-100 output
end divide_by_10;

architecture aaa of divide_by_10 is
  signal count : std_logic_vector(3 downto 0);

begin

  process (clk, rst_n)
  begin

    if rst_n = '0' then                 -- asynchronous reset (active low)
      count  <= "0000";                 -- reset count
      div100 <= '0';
    elsif clk'event and clk = '1' then  -- rising clock edge

      div100 <= '0';                    -- default to '0' output

      if count = 9 then                 -- at 100 counts?
        div100 <= '1';                  -- set output for one clock only
        count  <= "0000";               -- wrap to 0
      else
        count  <= count + 1;            -- increment count next clock
      end if;

    end if;

  end process;

end aaa;
```

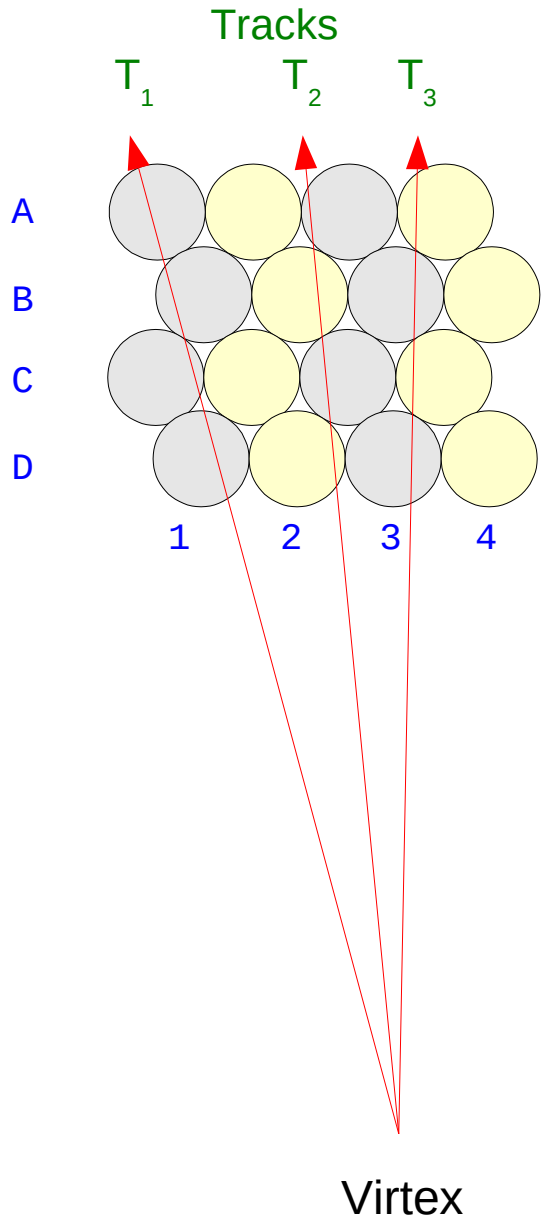
Signal declaration. Signals represent “wires” which connect elements together

Architecture Definition
an entity can have more than one alternative architecture

A process is a group of sequential statements controlled by one or more signals. Most processes are *synchronous* like this one, triggered by the rising edge of a clock.

These statements execute on every rising clock edge

Track Segment Finder Example



Here is an array of 16 circular detector elements (drift tubes or scintillating fibers)

Assume each is wired to the input of an FPGA, and we want to generate a trigger output for each case where a track segment is present.

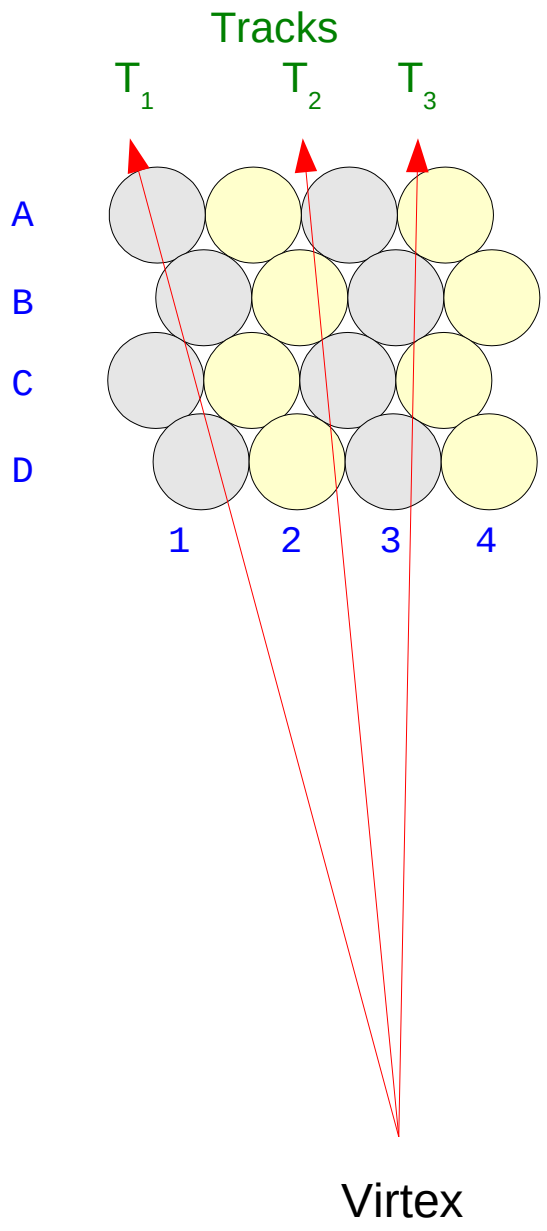
Here are boolean expressions for 3 example tracks:

```
T1 = A1 and B1 and C1 and D1
T2 = A3 and B2 and C3 and D3
T3 = A4 and B3 and C4 and D3
```

This is simple enough, but if we want to account for inefficiencies in the detector, then we must also include all forms with 1 channel missing:

```
T1 = (A1 and B1 and C1) or
     (A1 and B1 and D1) or
     (A1 and C1 and D1) or
     (B1 and C1 and D1)
```

Track Segment Finder



How do we generate the equations?

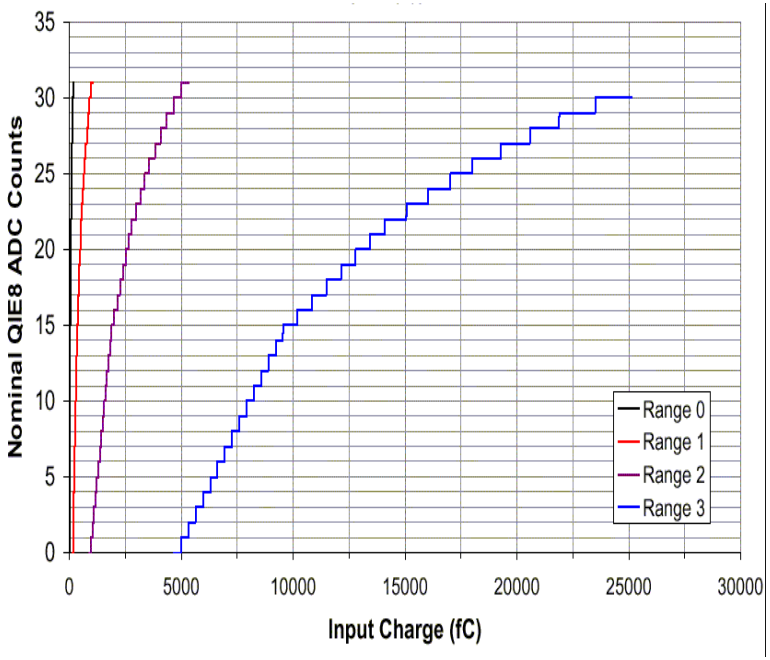
One option is to use geometry... for muon trackers where all the particles can be assumed to come from the interaction point this works well.

For central trackers, often a monte-carlo simulation of many events is used. Here is a single equation from the DØ Level 1 Central Track Trigger firmware!

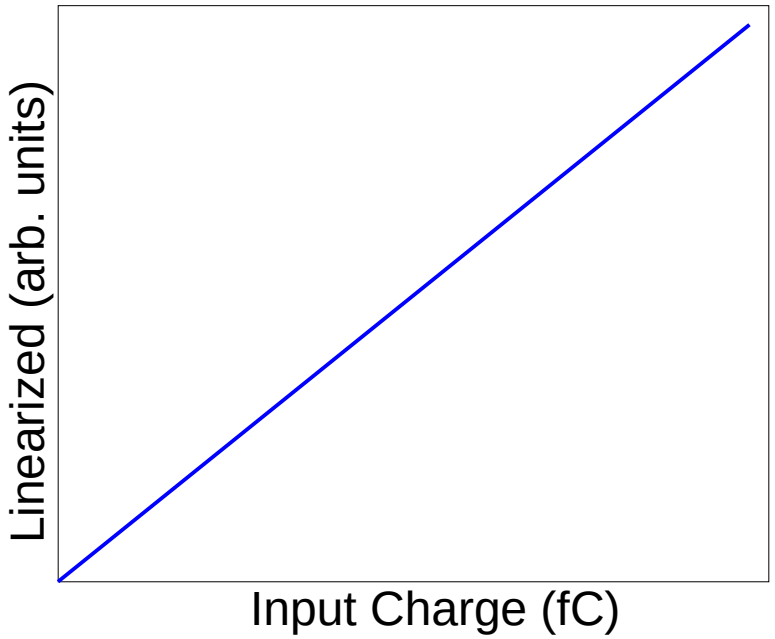
```
m(0) <= (ai(15) and bi(19) and ci(23) and co(23) and di(27) and do(28)
and eo(32) and fi(36) and gi(41) and go(41) and hi(45)) or
(ai(15) and bi(19) and bo(19) and co(23) and di(27) and eo(32) and fi(36) and fo(36) and go(41) and hi(45) and ho(45)) or
(ai(15) and bi(19) and ci(23) and di(27) and do(28) and eo(32) and fi(36) and gi(41) and hi(45) and ho(46)) or
(ai(15) and ao(15) and bo(19) and co(23) and di(27) and do(27) and ei(31) and eo(32) and fo(36) and go(41) and hi(45)) or
(ai(15) and bo(19) and co(23) and di(27) and ei(31) and eo(32) and fo(36) and go(41) and hi(45) and ho(45)) or
(ai(15) and ao(15) and bo(19) and co(23) and di(27) and do(27) and ei(31) and eo(32) and fo(36) and go(41) and ho(45)) or
(ai(15) and ao(15) and bo(19) and co(23) and di(27) and do(27) and ei(31) and fo(36) and gi(40) and go(41) and ho(45)) or
(ai(15) and ao(15) and bo(19) and co(23) and di(27) and do(27) and ei(31) and fo(36) and go(41) and hi(45) and ho(45)) or
(ai(15) and bi(19) and bo(19) and co(23) and di(27) and ei(31) and eo(32) and fo(36) and go(41) and ho(45)) or
(ai(15) and bo(19) and co(23) and di(27) and ei(31) and eo(32) and fo(36) and gi(40) and go(41) and ho(45)) or
(ai(15) and bo(19) and ci(23) and co(23) and di(27) and eo(32) and fi(36) and fo(36) and go(41) and hi(45) and ho(45)) or
(ai(15) and bo(19) and co(23) and di(27) and eo(32) and fi(36) and gi(41) and hi(45) and ho(46)) or
(ao(15) and bo(19) and ci(22) and co(23) and do(27) and ei(31) and fi(35) and fo(36) and gi(40) and go(41) and ho(45)) or
(ai(15) and ao(15) and bo(19) and co(23) and di(27) and ei(31) and eo(32) and fo(36) and gi(41) and go(41) and hi(45)) or
(ai(15) and bi(19) and ci(23) and co(23) and di(27) and eo(32) and fi(36) and fo(36) and gi(41) and go(41) and hi(45)) or
(ai(15) and bi(19) and bo(19) and ci(23) and co(23) and di(27) and eo(32) and fi(36) and fo(36) and go(41) and hi(45)) or
(ao(15) and bi(18) and bo(19) and ci(22) and co(23) and do(27) and ei(31) and fi(35) and fo(36) and gi(40) and ho(45)) or
(ao(15) and bi(18) and bo(19) and ci(22) and co(23) and do(27) and ei(31) and fo(36) and gi(40) and go(41) and ho(45)) or
(ai(15) and ao(16) and bi(19) and ci(23) and do(28) and eo(32) and fi(36) and gi(41) and go(42) and hi(45) and ho(46)) or
(ai(15) and ao(15) and bo(19) and co(23) and di(27) and ei(31) and eo(32) and fi(36) and fo(36) and go(41) and hi(45)) or
(ai(15) and bi(19) and bo(19) and ci(23) and di(27) and do(28) and eo(32) and fi(36) and fo(36) and gi(41) and go(41) and hi(45)) or
(ai(15) and bo(19) and co(23) and di(27) and eo(32) and fi(36) and fo(36) and gi(41) and go(41) and hi(45)) or
(ai(15) and bo(19) and ci(23) and co(23) and di(27) and eo(32) and fi(36) and gi(41) and go(41) and hi(45)) or
(ao(15) and bo(19) and co(23) and di(27) and do(27) and ei(31) and eo(32) and fi(36) and fo(36) and gi(41) and go(41) and hi(45)) or
(ao(15) and bi(18) and bo(19) and ci(22) and do(27) and ei(31) and eo(31) and fi(35) and gi(40) and go(41) and ho(45)) or
(ao(15) and bi(18) and bo(19) and ci(22) and co(23) and do(27) and ei(31) and fi(35) and fo(36) and go(41) and ho(45)) or
(ai(15) and bi(19) and bo(19) and ci(23) and co(23) and di(27) and do(28) and eo(32) and fi(36) and gi(41) and hi(45)) or
(ao(15) and bi(18) and ci(22) and di(26) and do(27) and ei(31) and eo(31) and fi(35) and gi(40) and ho(45)) or
(ao(15) and bo(19) and ci(22) and co(23) and di(27) and do(27) and ei(31) and fo(36) and go(41) and hi(45) and ho(45)) or
(ao(15) and bo(19) and co(23) and di(27) and do(27) and ei(31) and eo(32) and fo(36) and go(41) and hi(45) and ho(45));
```

Calorimetry Example - Linearizing Energy Scale

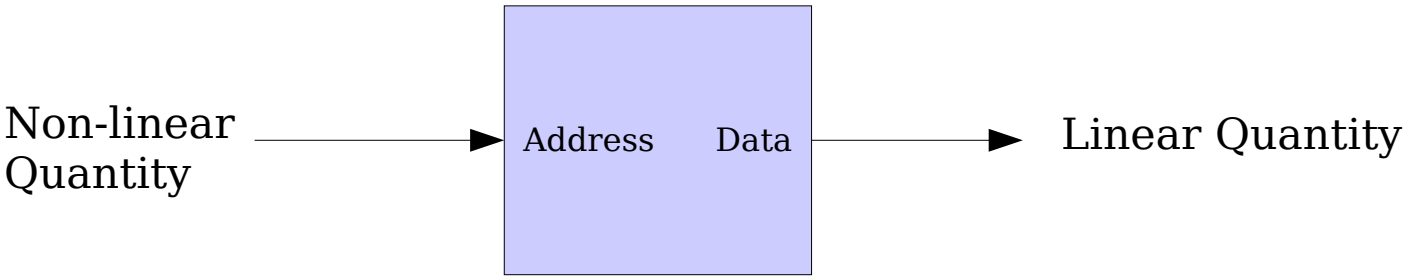
Non-Linear ADC scale (i.e. QIE)



Linear Energy Scale
(necessary for summing)



In an FPGA, this is typically done with a look-up table (LUT).
A LUT is just a memory where the address is the non-linear value, while the data stored at that address is the linear value.



Calorimetry Example - Linearizing Energy Scale

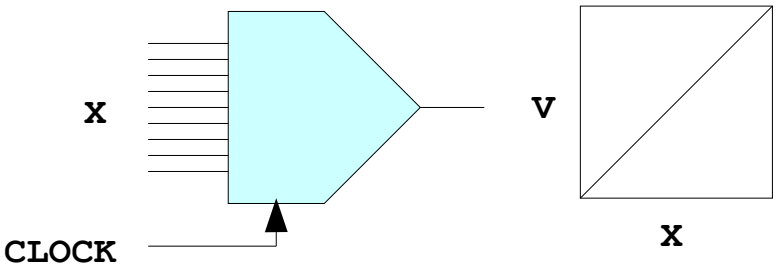
- Nonlinear ADC with 7-bit output (i.e. QIE) requires 128 RAM cells.
- You can also perform any needed calibration, pedestal subtracion, etc in a LUT.
- A large FPGA has ~100 block RAMs, each 18k bits (each could hold 8 128x16 LUTs)

```
entity linearize_example is

  port (
    clock           : in  std_logic;
    nonlinear_value : in  std_logic_vector(6 downto 0);
    linear_value    : out std_logic_vector(17 downto 0));

end linearize_example;

bram1 : RAMB16_S18
  generic map (
    -- table of values
    INIT_00 => X"abadcafeabadcafeabadcafeabadcafeabadcafeabadcafeabadcafeabadcafe",
    INIT_01 => X"deadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef",

    INIT_3E => X"feedbabefeedbabefeedbabefeedbabefeedbabefeedbabefeedbabefeedbabe",
    INIT_3F => X"0123456701234567012345670123456701234567012345670123456701234567",
    port map (
      DO    => linear_value,          -- 16-bit Data Output
      ADDR  => nonlinear_value,       -- 10-bit Address Input
      CLK   => clock,                 -- Clock
      EN    => '1'                    -- RAM Enable Input
      );
```

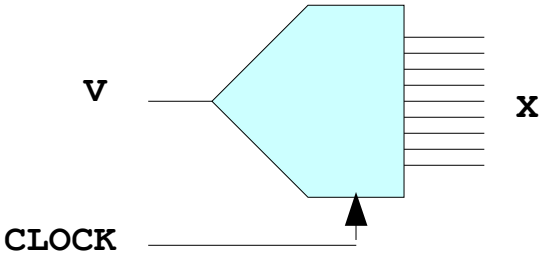
This is an example of a pre-defined block (RAM, in this case) *instantiated* in a design file

Some Other Building Blocks

- DAC (Digital to Analog Converter)
  Converts a binary value to a voltage



- ADC (Analog to Digital Converter)
  Converts a voltage to a binary value



ADCs and DACs convert one sample per clock cycle. The maximum clock rate is called the *sampling rate.*

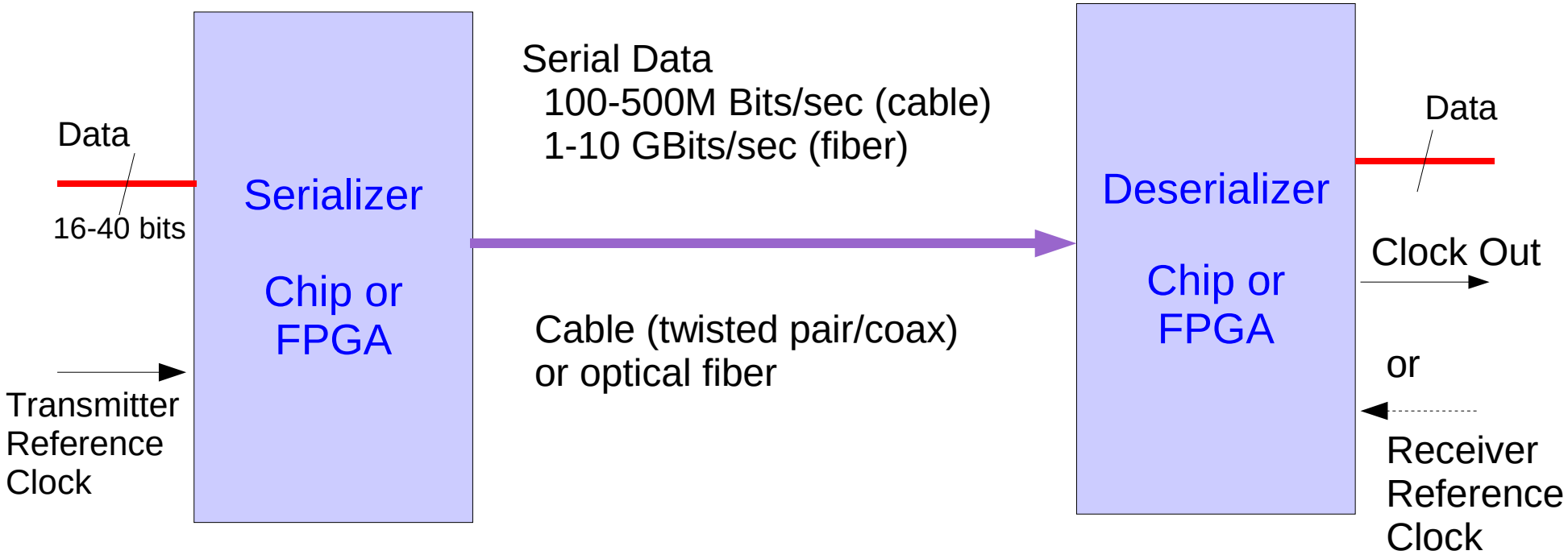
For HEP applications, ADC which sample at the accelerator RF clock (40MHz – CERN and 53MHz – FNAL) are quite popular.

Data Links

- Optical Fiber links
  - Relatively expensive
  - Used for long distance runs (>10m up to many km)
  - Fast (today: 1-10GB/sec, soon: 10-40GB/sec)
- Copper links (circuit board traces or cables)
  - LVDS Serial links
    - Built-in to modern FPGAs, so “free”
    - Up to 10GB/sec for short runs, 500MB/sec for 10-20m runs
  - Other technologies (Hot-Link, Taxi, Vitesse...)
    - Built-in equalization to compensate for dispersion in long cables
    - Becoming obsolete, but still widely used

Data Links

- To the user, links essentially all look the same:



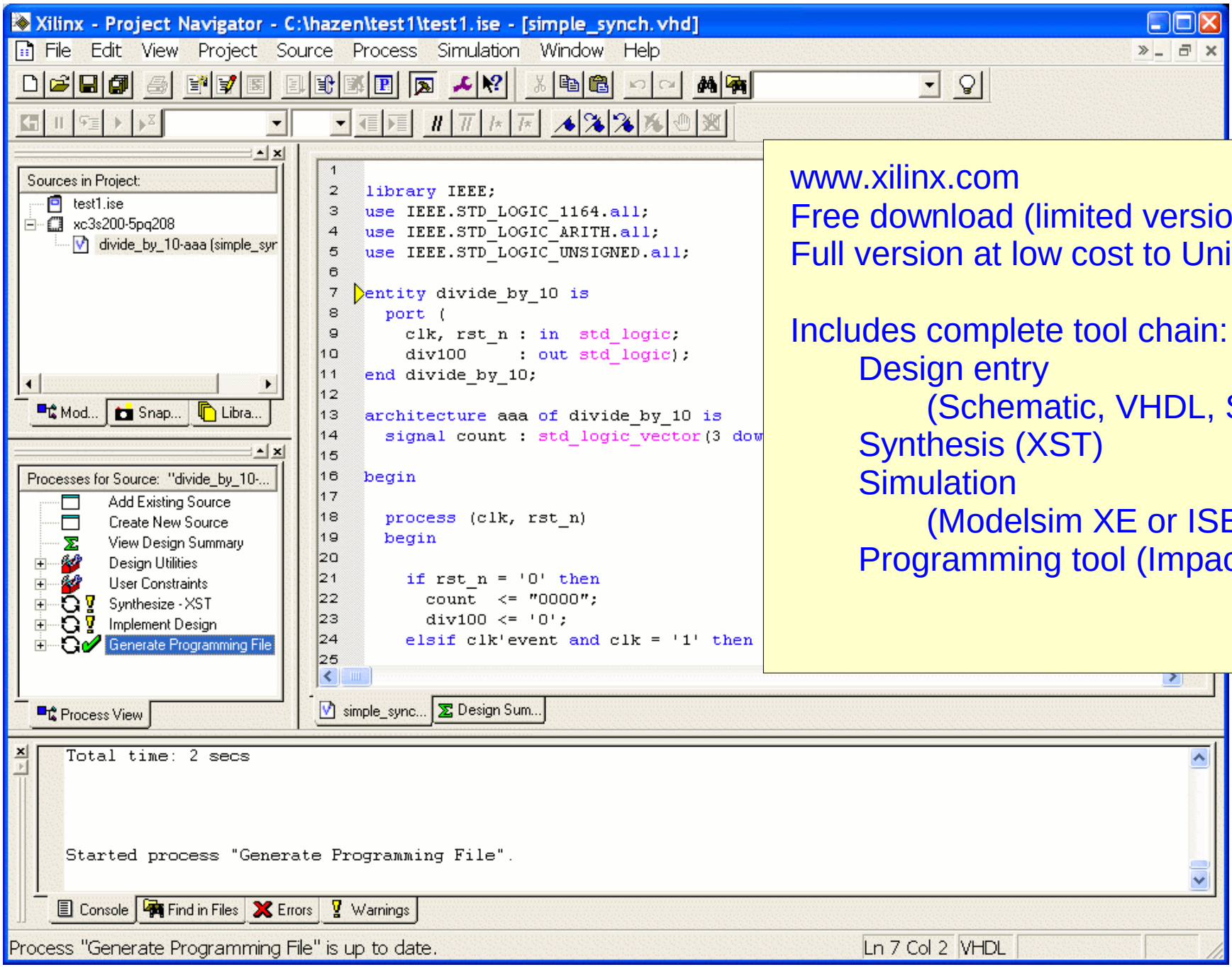
Transmitter accepts a clock (constant rate) and one parallel data word per clock cycle

Data appears at receiver after a certain latency delay (in addition to cable/fiber propagation delay)

FPGA Design Tools

- For FPGA logic design, you need some software:
  - An editor. Emacs is fine, and has a very nice VHDL mode
  - A *synthesis* tool, which translates VHDL into RTL
    - (Register Transfer Logic, essentially boolean expressions and registers)
    - This tool can come from the FPGA vendor (Xilinx, Altera) or may be a separate program (i.e. Synplify, Leonardo Spectrum)
  - Implementation tools, which convert the generic RTL into a file which can be downloaded to the FPGA
    - These are always provided by the FPGA vendor
  - A simulator. There are two types of simulation:
    - *Functional Simulation* – based only on VHDL. Tests for correct design.
    - *Timing Simulation* – uses post-implementation data, check for timing problems.
    - Simulator may be supplied by the FPGA vendor, or a separate tool

Xilinx ISE 7.1



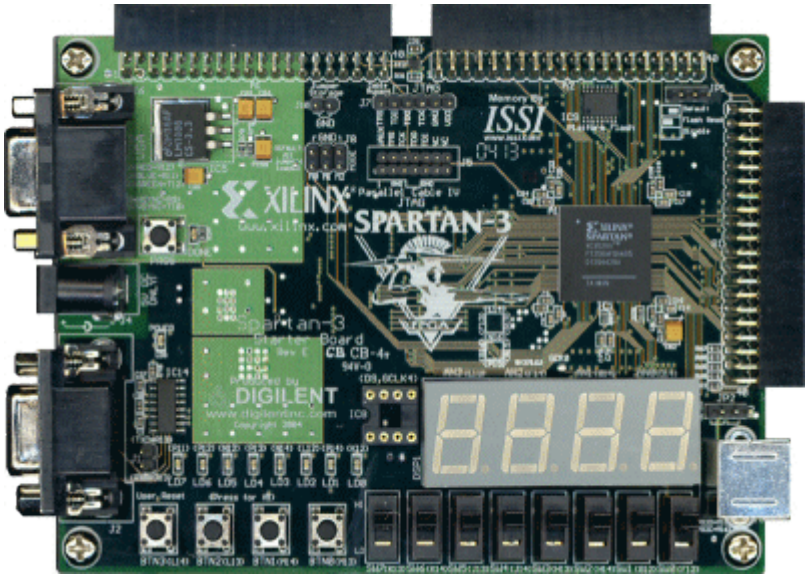
www.xilinx.com
Free download (limited version)
Full version at low cost to Universities

Includes complete tool chain:

- Design entry
  - (Schematic, VHDL, StateCAD)
- Synthesis (XST)
- Simulation
  - (Modelsim XE or ISE Simulator)
- Programming tool (Impact)

Getting Started

- Download free software
- Buy an Evaluation Board
  for example:
  - $99 Spartan-3 Board from Digilent
    - Comes with all cables, etc
      so you can get started immediately
    - Uses Spartan-3 FPGA
      - 90nm CMOS, latest technology
    - Has a serial port and RAM so
      you can experiment with the
      Microblaze embedded CPU
    - See www.digilentinc.com



Summary

- Trigger systems for large colliders are very complex systems, but are built of simple building blocks
- Large FPGAs and modern design tools have substantially reduced the learning curve required for logic design.
- Careful engineering is still needed for the interfaces.
- Electronics is lots of fun, and we couldn't do it without huge contributions made by you! (Students – Grad and Undergrad, Post-docs and even faculty)