Technical University of Denmark

Exercise 1 - Ethernet FCS

Course:
34349: FPGA design for Communication systems E15

Author(s):

Alexander Brandi (s103208)
Mads Emil Kragelund (s134530)
Martin Jesper Low Madsen (s124320)
Christian Kjær Laustsen (s124324)

30. September 2015

Table of Contents

1 Introduction

This exercise will go through the design of a bit error checker for an Ethernet frame where the last 4 octets in the frame is the *frame check sequence*, FCS, which is shown in figure 1.

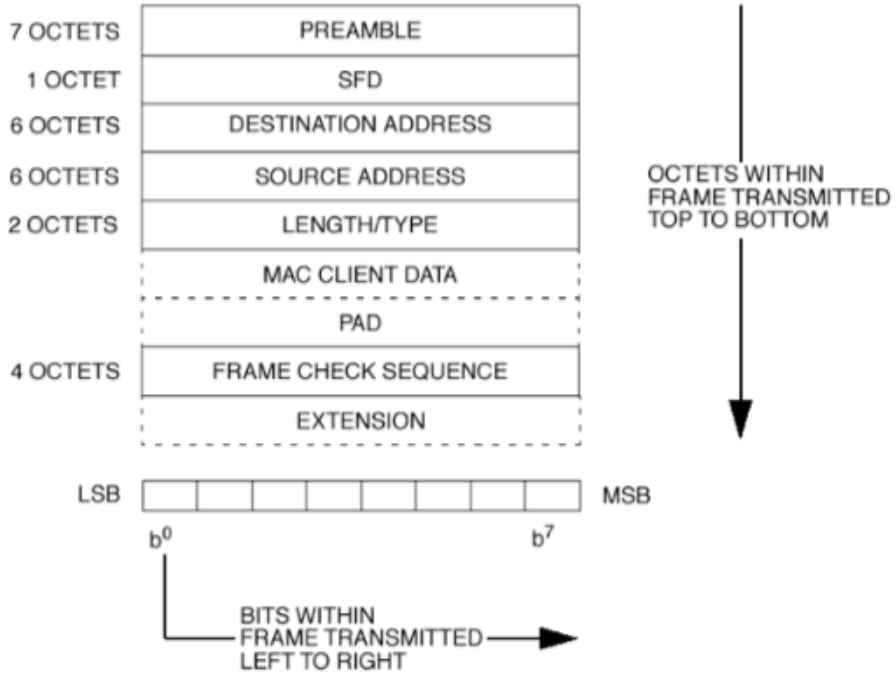

Figure 1: Ethernet frame

To generate a value from the FCS field, a *cyclic redundancy check*, CRC, is used meaning the FCS field contains a 32-bit CRC value. The encoding of the CRC value is defined as such,

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1 \quad \text{(1a)}$$

To decode the value, the 32-bit FCS is fed into a shift-register which starts filled with 0's. If the shift-register only contains 0's at the end, then the data is error free.

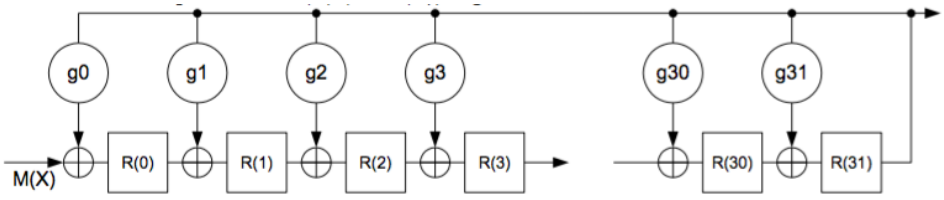

Figure 2: Shift-register calculating the CRC value

2 Serial Implementation

First a module will be constructed that checks the Ethernet packet for bit errors, assuming that the packet is received as a serial stream. Alternatively, a Python implementation of the algorithm can be found in appendix A.

The following is the full VHDL code for the module *fcs_check_serial*.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity fcs_check_serial is
  port (clk            : in  std_logic;
        reset          : in  std_logic;
        start_of_frame : in  std_logic;
        end_of_frame   : in  std_logic;  -- Arrival of the first bit in CRC checksum.
        data_in        : in  std_logic;
        fcs_error      : out std_logic);
end fcs_check_serial;

architecture behavioral of fcs_check_serial is
  -- Remainder shift register
  signal R           : std_logic_vector(31 downto 0);

  signal shift_count : unsigned(4 downto 0);
  signal data        : std_logic;
  signal fcs_done    : std_logic;
begin

  -- Complement the first and last 32 bits of the frame by waiting for the
  -- first and last bit of the frame or by ensuring an appropriate amount of
  -- shifts are performed.
  process (shift_count, start_of_frame, end_of_frame, data_in)
  begin
    data <= data_in;

    if shift_count < 31 or start_of_frame = '1' or end_of_frame = '1' then
      data <= not data_in;
    end if;
  end process;

  -- Serial linear feedback shift register logic.
  process (clk, reset)
  begin
    if reset = '1' then
      R           <= (others => '0');
```

```
    shift_count <= (others => '0');
    fcs_error   <= '1';
  elsif rising_edge(clk) then
    if end_of_frame = '1' then
      fcs_done <= '1';
    end if;

    if start_of_frame = '1' or end_of_frame = '1' then
      shift_count <= (others => '0');
    elsif shift_count < 31 then
      shift_count <= shift_count + 1;
    end if;

    -- Serial linear feedback shift registers:
    --
    -- The generator polynomial is given by:
    -- 1 0000010 011000001 00011101 10110111
    -- MSB                                 LSB
    R(0) <= data xor R(31);           -- x^0
    R(1) <= R(0) xor R(31);           -- x^1
    R(2) <= R(1) xor R(31);           -- x^2
    R(3) <= R(2);
    R(4) <= R(3) xor R(31);           -- x^4
    R(5) <= R(4) xor R(31);           -- x^5
    R(6) <= R(5);
    R(7) <= R(6) xor R(31);           -- x^7

    R(8)  <= R(7) xor R(31);          -- x^8
    R(9)  <= R(8);
    R(10) <= R(9) xor R(31);          -- x^10
    R(11) <= R(10) xor R(31);         -- x^11
    R(12) <= R(11) xor R(31);         -- x^12
    R(13) <= R(12);
    R(14) <= R(13);
    R(15) <= R(14);

    R(16) <= R(15) xor R(31);         -- x^16
    R(17) <= R(16);
    R(18) <= R(17);
    R(19) <= R(18);
    R(20) <= R(19);
    R(21) <= R(20);
    R(22) <= R(21) xor R(31);         -- x^22
    R(23) <= R(22) xor R(31);         -- x^23
    R(24) <= R(23);
```

```
      R(25) <= R(24);
      R(26) <= R(25) xor R(31);          -- x^26
      R(27) <= R(26);
      R(28) <= R(27);
      R(29) <= R(28);
      R(30) <= R(29);
      R(31) <= R(30);
      -- R(31) is x^32

      if R = "00000000000000000000000000000000" and fcs_done = '1' then
        fcs_error <= '0';
      end if;
    end if;
  end process;

end behavioral;
```

3 Synthesis and Place & Route

First the synthesis of the *fcs_check_serial* entity is performed, as shown in figure 3.

Analysis & Synthesis Summary	
Analysis & Synthesis Status	Successful - Wed Sep 30 22:51:50 2015
Quartus II 64-Bit Version	15.0.0 Build 145 04/22/2015 SJ Web Edition
Revision Name	test
Top-level Entity Name	fcs_check_serial
Family	Cyclone V
Logic utilization (in ALMs)	N/A
Total registers	38
Total pins	6
Total virtual pins	0
Total block memory bits	0
Total DSP Blocks	0
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0
Total DLLs	0

Figure 3: Synthesis of *fcs_check_serial*

Then a Place & Route is performed on *fcs_check_serial* entity is performed, as shown in figure 4.

Fitter Summary	
Fitter Status	Successful - Wed Sep 30 22:53:10 2015
Quartus II 64-Bit Version	15.0.0 Build 145 04/22/2015 SJ Web Edition
Revision Name	test
Top-level Entity Name	fcs_check_serial
Family	Cyclone V
Device	5CGXFC7C7F23C8
Timing Models	Final
Logic utilization (in ALMs)	19 / 56,480 (< 1 %)
Total registers	45
Total pins	6 / 268 (2 %)
Total virtual pins	0
Total block memory bits	0 / 7,024,640 (0 %)
Total RAM Blocks	0 / 686 (0 %)
Total DSP Blocks	0 / 156 (0 %)
Total HSSI RX PCSs	0 / 6 (0 %)
Total HSSI PMA RX Deserializers	0 / 6 (0 %)
Total HSSI TX PCSs	0 / 6 (0 %)
Total HSSI PMA TX Serializers	0 / 6 (0 %)
Total PLLs	0 / 13 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 4: Place & Route of *fcs_check_serial*

From the timing analyzer, the speed is noted to be 360.88 MHz, as shown in figure 5.

Slow 1100mV 85C Model Fmax Summary				
	Fmax	Restricted Fmax	Clock Name	Note
1	360.88 MHz	360.88 MHz	clk	

Figure 5: Timing Analysis of *fcs_check_serial*

4 Testbench for Serial Implementation

To test the *fcs_check_serial* module without having to manually input everything, a testbench, *fcs_check_serial_test*, is constructed. The testbench uses the *fcs_check_serial* entity, and sends

signals to it.

The following is the *fcs_check_serial_test* testbench.

```
library ieee;
use ieee.std_logic_1164.all;

entity fcs_check_serial_test is
end fcs_check_serial_test;

architecture behavior of fcs_check_serial_test is

  component fcs_check_serial
    port(clk            : in  std_logic;
         reset          : in  std_logic;
         start_of_frame : in  std_logic;
         end_of_frame   : in  std_logic;
         data_in        : in  std_logic;
         fcs_error      : out std_logic);
  end component;

  -- Inputs
  signal clk            : std_logic := '0';
  signal reset          : std_logic := '0';
  signal start_of_frame : std_logic := '0';
  signal end_of_frame   : std_logic := '0';
  signal data_in        : std_logic := '0';

  -- Outputs
  signal fcs_error : std_logic;

  -- Clock period definitions
  constant clk_period : time := 10 ns;

  -- Data to feed the FCS check entity.
  constant data_to_send_in : std_logic_vector(511 downto 0) :=
    x"00_10_A4_7B_EA_80_00_12_34_56_78_90_08_00_45_00_00_2E_B3_FE_00_00_80_11" &
    x"05_40_C0_A8_00_2C_C0_A8_00_04_04_00_04_00_00_1A_2D_E8_00_01_02_03_04_05" &
    x"06_07_08_09_0A_0B_0C_0D_0E_0F_10_11_E6_C5_3D_B2";

begin

  i_fcs_check_1 : fcs_check_serial port map (
    clk            => clk,
    reset          => reset,
    start_of_frame => start_of_frame,
    end_of_frame   => end_of_frame,
```

```
    data_in         => data_in,
    fcs_error       => fcs_error
    );

  -- Test clock.
  clk_process : process
  begin
    clk <= '0';
    wait for clk_period / 2;
    clk <= '1';
    wait for clk_period / 2;
  end process;

  -- Stimulus process.
  stim_proc : process
  begin
    -- Reset the state.
    reset <= '1';
    wait for clk_period;
    reset <= '0';

    -- Start sending data, and indicate we are no longer at the start of the
    -- frame.
    for i in data_to_send_in'range loop
      if i = (data_to_send_in'length - 1) then
        start_of_frame <= '1';
      else
        start_of_frame <= '0';
      end if;

      data_in <= data_to_send_in(i);

      -- Start of last frame (32 bits).
      if i = 31 then
        end_of_frame <= '1';
      else
        end_of_frame <= '0';
      end if;

      wait for clk_period;
    end loop;

    data_in <= '0';

    wait;
  end process;
```

```
end;
```

5 Parallel Implementation

Finally, a parallel implementation of the *fcs_check_serial* is made, called *fcs_check_parallel*. This will take in 8-bit octets at a time, instead of consuming the data serially.

The following is the implementation of *fcs_check_parallel*.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity fcs_check_parallel is
  port (clk           : in  std_logic;
        reset         : in  std_logic;
        start_of_frame : in  std_logic;
        end_of_frame   : in  std_logic;  -- Arrival of the first byte in CRC checksum.
        data_in       : in  std_logic_vector(7 downto 0);
        fcs_error     : out std_logic);
end fcs_check_parallel;

architecture behavioral of fcs_check_parallel is
  -- Remainder shift register
  signal R           : std_logic_vector(31 downto 0);

  signal shift_count : unsigned(1 downto 0);
  signal data        : std_logic_vector(7 downto 0);
  signal fcs_done    : std_logic;
begin

  -- Complement the first and last 32 bits of the frame by waiting for the
  -- first and last bit of the frame or by ensuring an appropriate amount of
  -- shifts are performed.
  process (shift_count, start_of_frame, end_of_frame, data_in)
  begin
    data <= data_in;

    if shift_count < 3 or start_of_frame = '1' or end_of_frame = '1' then
      data <= not data_in;
    end if;
  end process;
```

```
-- 8-bit parallel linear feedback shift register logic based on a shift
-- matrix.
process (clk, reset)
begin
  if reset = '1' then
    R           <= (others => '0');
    shift_count <= (others => '0');
    fcs_error   <= '1';
    fcs_done    <= '0';
  elsif rising_edge(clk) then
    if end_of_frame = '1' then
      fcs_done <= '1';
    end if;

    if start_of_frame = '1' or end_of_frame = '1' then
      shift_count <= (others => '0');
    elsif shift_count < 3 then
      shift_count <= shift_count + 1;
    end if;

    R(0) <= R(24) xor R(30) xor data(0);
    R(1) <= R(24) xor R(25) xor R(30) xor R(31) xor data(1);
    R(2) <= R(24) xor R(25) xor R(26) xor R(30) xor R(31) xor data(2);
    R(3) <= R(25) xor R(26) xor R(27) xor R(31) xor data(3);
    R(4) <= R(24) xor R(26) xor R(27) xor R(28) xor R(30) xor data(4);
    R(5) <= R(24) xor R(25) xor R(27) xor R(28) xor R(29) xor R(30) xor
            R(31) xor data(5);
    R(6) <= R(25) xor R(26) xor R(28) xor R(29) xor R(30) xor R(31) xor
            data(6);
    R(7)  <= R(24) xor R(26) xor R(27) xor R(29) xor R(31) xor data(7);
    R(8)  <= R(0) xor R(24) xor R(25) xor R(27) xor R(28);
    R(9)  <= R(1) xor R(25) xor R(26) xor R(28) xor R(29);
    R(10) <= R(2) xor R(24) xor R(26) xor R(27) xor R(29);
    R(11) <= R(3) xor R(24) xor R(25) xor R(27) xor R(28);
    R(12) <= R(4) xor R(24) xor R(25) xor R(26) xor R(28) xor R(29) xor
             R(30);
    R(13) <= R(5) xor R(25) xor R(26) xor R(27) xor R(29) xor R(30) xor
             R(31);
    R(14) <= R(6) xor R(26) xor R(27) xor R(28) xor R(30) xor R(31);
    R(15) <= R(7) xor R(27) xor R(28) xor R(29) xor R(31);
    R(16) <= R(8) xor R(24) xor R(28) xor R(29);
    R(17) <= R(9) xor R(25) xor R(29) xor R(30);
    R(18) <= R(10) xor R(26) xor R(30) xor R(31);
    R(19) <= R(11) xor R(27) xor R(31);
    R(20) <= R(12) xor R(28);
    R(21) <= R(13) xor R(29);
```

```
      R(22) <= R(14) xor R(24);
      R(23) <= R(15) xor R(24) xor R(25) xor R(30);
      R(24) <= R(16) xor R(25) xor R(26) xor R(31);
      R(25) <= R(17) xor R(26) xor R(27);
      R(26) <= R(18) xor R(24) xor R(27) xor R(28) xor R(30);
      R(27) <= R(19) xor R(25) xor R(28) xor R(29) xor R(31);
      R(28) <= R(20) xor R(26) xor R(29) xor R(30);
      R(29) <= R(21) xor R(27) xor R(30) xor R(31);
      R(30) <= R(22) xor R(28) xor R(31);
      R(31) <= R(23) xor R(29);

      if R = "00000000000000000000000000000000" and fcs_done = '1' then
        fcs_error <= '0';
      end if;
    end if;
  end process;

end behavioral;
```

The above VHDL code uses complex boolean expressions to determine the values of the registers for each shift. These boolean expressions are derived from a shift matrix raised to the power of eight, the number of bits. A Python implementation of such matrix is found in appendix B.

5.1 Parallel testbench

As with the serial implementation, a testbench, *fcs_check_parallel_test*, is created to test the parallel implementation.

The following is the implementation of the *fcs_check_parallel_test* entity.

```
library ieee;
use ieee.std_logic_1164.all;

entity fcs_check_parallel_test is
end fcs_check_parallel_test;

architecture behavior of fcs_check_parallel_test is
  component fcs_check_parallel
    port(clk           : in  std_logic;
         reset         : in  std_logic;
         start_of_frame : in  std_logic;
         end_of_frame   : in  std_logic;
         data_in       : in  std_logic_vector(7 downto 0);
         fcs_error     : out std_logic);
  end component;
```

```
  -- Inputs.
  signal clk            : std_logic                     := '0';
  signal reset          : std_logic                     := '0';
  signal start_of_frame : std_logic                     := '0';
  signal end_of_frame   : std_logic                     := '0';
  signal data_in        : std_logic_vector(7 downto 0) := (others => '0');

  -- Outputs.
  signal fcs_error : std_logic;

  -- Clock period definitions.
  constant clk_period      : time := 10 ns;

  -- Data to feed the FCS check entity.
  constant data_to_send_in : std_logic_vector(511 downto 0) :=
    x"00_10_A4_7B_EA_80_00_12_34_56_78_90_08_00_45_00_00_2E_B3_FE_00_00_80_11" &
    x"05_40_C0_A8_00_2C_C0_A8_00_04_04_00_04_00_00_1A_2D_E8_00_01_02_03_04_05" &
    x"06_07_08_09_0A_0B_0C_0D_0E_0F_10_11_E6_C5_3D_B2";

begin

  i_fcs_check_1 : fcs_check_parallel
    port map (clk           => clk,
              reset         => reset,
              start_of_frame => start_of_frame,
              end_of_frame  => end_of_frame,
              data_in       => data_in,
              fcs_error     => fcs_error);

  -- Test clock.
  clk_process : process
  begin
    clk <= '0';
    wait for clk_period / 2;
    clk <= '1';
    wait for clk_period / 2;
  end process;


  -- Stimulus process.
  stim_proc : process
  begin
    -- Reset the entity
    reset <= '1';

    wait for clk_period;
```

```
    reset <= '0';

    -- Start sending data, and indicate we are no longer at the start of the
    -- frame.
    for i in ((data_to_send_in'length / 8) - 1) downto 0 loop
      if i = (data_to_send_in'length - 1) then
        start_of_frame <= '1';
      else
        start_of_frame <= '0';
      end if;

      data_in <= data_to_send_in(((i + 1) * 8 - 1) downto i * 8);

      -- Start of last frame (32 bits).
      if i = 3 then
        end_of_frame <= '1';
      else
        end_of_frame <= '0';
      end if;

      wait for clk_period;
    end loop;

    data_in <= "00000000";

    wait;
  end process;

end;
```

5.2 Synthesis and Place & Route

First the synthesis of the *fcs_check_parallel* entity is performed, as shown in figure 6.

Analysis & Synthesis Summary	
Analysis & Synthesis Status	Successful - Wed Sep 30 23:00:04 2015
Quartus II 64-Bit Version	15.0.0 Build 145 04/22/2015 SJ Web Edition
Revision Name	test
Top-level Entity Name	fcs_check_parallel
Family	Cyclone V
Logic utilization (in ALMs)	N/A
Total registers	36
Total pins	13
Total virtual pins	0
Total block memory bits	0
Total DSP Blocks	0
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0
Total DLLs	0

Total block memory bits

Figure 6: Synthesis of *fcs_check_parallel*

Then a Place & Route is performed on *fcs_check_parallel* entity is performed, as shown in figure 7.

Fitter Summary	
Fitter Status	Successful - Wed Sep 30 23:01:18 2015
Quartus II 64-Bit Version	15.0.0 Build 145 04/22/2015 SJ Web Edition
Revision Name	test
Top-level Entity Name	fcs_check_parallel
Family	Cyclone V
Device	5CGXFC7C7F23C8
Timing Models	Final
Logic utilization (in ALMs)	30 / 56,480 (< 1 %)
Total registers	50
Total pins	13 / 268 (5 %)
Total virtual pins	0
Total block memory bits	0 / 7,024,640 (0 %)
Total RAM Blocks	0 / 686 (0 %)
Total DSP Blocks	0 / 156 (0 %)
Total HSSI RX PCSs	0 / 6 (0 %)
Total HSSI PMA RX Deserializers	0 / 6 (0 %)
Total HSSI TX PCSs	0 / 6 (0 %)
Total HSSI PMA TX Serializers	0 / 6 (0 %)
Total PLLs	0 / 13 (0 %)
Total DLLs	0 / 4 (0 %)

Top-level Entity Name

Figure 7: Place & Route of *fcs_check_parallel*

From the timing analyzer, the speed is noted to be 293 MHz, as shown in figure 8.

Slow 1100mV 85C Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	293.0 MHz	293.0 MHz	clk	

Figure 8: Timing Analysis of *fcs_check_parallel*

Meaning the parallel implementation is $\frac{360.88MHz}{293MHz} = 1.231672355$ times faster than the serial implementation.

A Initial Python implementation

```
G = [1,
     0, 0, 0, 0, 0, 1, 0, 0,
     1, 1, 0, 0, 0, 0, 0, 1,
     0, 0, 0, 1, 1, 1, 0, 1,
     1, 0, 1, 1, 0, 1, 1, 1]

degree = 32

frameData = [0x00, 0x10, 0xA4, 0x7B,
             0xEA, 0x80, 0x00, 0x12,
             0x34, 0x56, 0x78, 0x90,
             0x08, 0x00, 0x45, 0x00,
             0x00, 0x2E, 0xB3, 0xFE,
             0x00, 0x00, 0x80, 0x11,
             0x05, 0x40, 0xC0, 0xA8,
             0x00, 0x2C, 0xC0, 0xA8,
             0x00, 0x04, 0x04, 0x00,
             0x04, 0x00, 0x00, 0x1A,
             0x2D, 0xE8, 0x00, 0x01,
             0x02, 0x03, 0x04, 0x05,
             0x06, 0x07, 0x08, 0x09,
             0x0A, 0x0B, 0x0C, 0x0D,
             0x0E, 0x0F, 0x10, 0x11]

# Convert hex values to a binary list.
binarySequence = []
for hexValue in frameData:
    binaryValue = '{:08b}'.format(hexValue)
    binarySequence = binarySequence + [int(x) for x in binaryValue]

# Complement first 32 bits of frame
M = [(x + 1) % 2 for x in binarySequence[0:degree]]

# Concatenate the original bits with the complemented ones
M = M + binarySequence[degree:]

# Append 32 zeroes
M = M + [0 for x in range(degree)]

# Iterate through T (copy of M) and perform a bitwise XOR with generator polynomium of
for i in range(len(M)):
    if M[i] and (len(M) - i) > degree:
        for j in range(len(G)):
            M[i + j] = M[i + j] ^^ G[j]
```

```
M = [(x + 1) % 2 for x in M]

# Fetch the remainder.
R = M[-32:]

# Actual calculated output
R[0:8]
R[8:16]
R[16:24]
R[24:32]

# Expected output
'{:08b}'.format(0xE6)
'{:08b}'.format(0xC5)
'{:08b}'.format(0x3D)
'{:08b}'.format(0xB2)
```

B Parallel matrix generation

```
# Define 'mod 2' environment
Mod2 = IntegerModRing(2)

# Parallel matrix generation
G = [#1,
     0, 0, 0, 0, 0, 1, 0, 0,
     1, 1, 0, 0, 0, 0, 0, 1,
     0, 0, 0, 1, 1, 1, 0, 1,
     1, 0, 1, 1, 0, 1, 1, 1]
zero = column_matrix(zero_vector(31))
identity = matrix.identity(31)

# Upper left part of the matrix
upperLeft = zero.augment(identity)
upperLeft = upperLeft.stack(matrix(G[::-1]))

# Upper right part of the matrix
upperRight = matrix(32, 8, 0)

# Bottom right part of the matrix
bottomRight = matrix(zero_vector(7)).stack(matrix.identity(7))
bottomRight = bottomRight.augment(zero_vector(8))
bottomRight[0,7] = 1

# Bottom left part of the matrix
bottomLeft = matrix(8, 32, 0)
bottomLeft[0,0] = 1

# Upper part of the matrix
top = upperLeft.augment(upperRight)

# Bottom part of the matrix
bottom = bottomLeft.augment(bottomRight)

# Full matrix
A = top.stack(bottom)
A = matrix(Mod2, A)

# Raise the A matrix to the power of 8
A8 = A^8

# Iterate through the matrix and find the boolean expression for each register.
for x in range(A8.ncols()):
    if x > 31:
```

```
        break
    Rx = []
    print 'R({0}) <= '.format(x),
    for y in range(A8[:, x].nrows()):
        if A8[y, x]:
            if y > 31:
                Rx.append('data_in({0})'.format(y - 32))
            else:
                Rx.append('R({0})'.format(y))
    print '{0};'.format(' xor '.join(Rx))

print A8.str(zero='.')
```