Imperial College London

Lecture 14

FPGA Embedded Memory

Peter Cheung
Department of Electrical & Electronic Engineering
Imperial College London

URL: www.ee.imperial.ac.uk/pcheung/teaching/ee2_digital/
E-mail: p.cheung@imperial.ac.uk

Lecture Objectives

- Different ways to use memory inside FPGAs
  - Register File
  - ROM and waveform generation
  - First-in-First-Out memory
- Memory resources inside Cyclone V FPGAs
- M9K memory block
- Library components in Quartus (IP Catalog)

References:

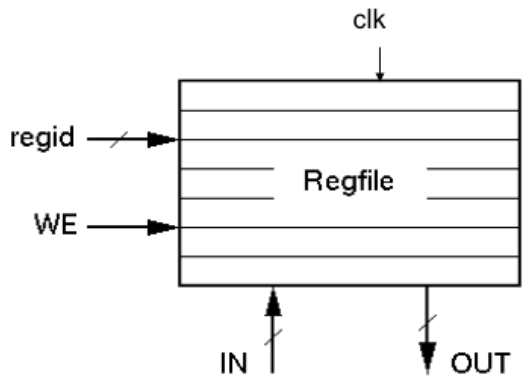
- "Cyclone V Device Handbook, Vol. 1"
- "Megafunction Overview User Guide"
- "RAM based shift register Megafunction User Guide"
- "SCFIFO and DCFIFO Megafunction User Guide"

In this lecture, we will consider the various type of storage (memory) that FPGAs allow us to implement. The major advantage of FPGAs is that it contains lots of small blocks of memory modules, which can either be used independently, or combined to form larger memory blocks. They also provide various configurations such as multi-port or registered input/output for data and address.

There are various useful references you can look up if you are interested to learn more about this. For the purpose of examination, the contents in this lecture and in the VERI experiment are sufficient.

Register File

- Register file from microprocessor



regid = register identifier (address of word in memory)
sizeof(regid) = log2(# of reg)
WE = write enable

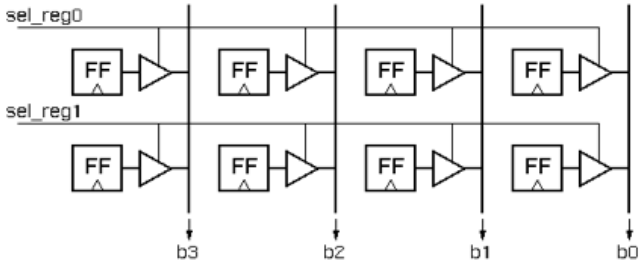
The simplest form of storage is a register file. All microprocessors have register files, which are known as “registers” in the architectural context.

Register files are fast, large and flexible. They are generally used to store temporary data for easy access by the ALU or floating point unit of a microprocessor, or for computational engine of a application specify digital system.

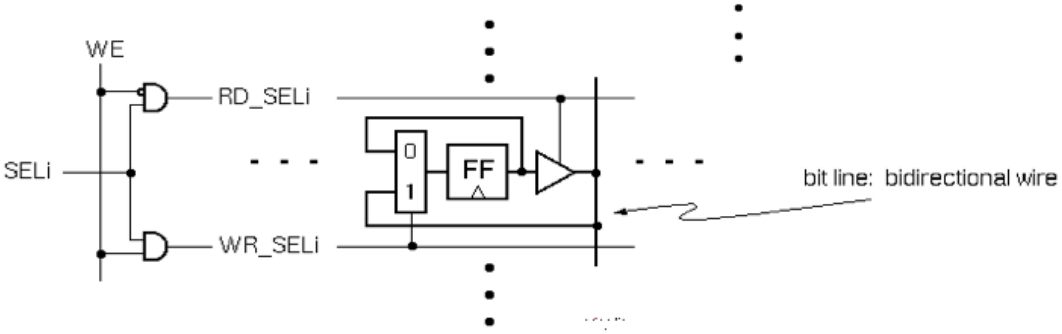
On the FPGA, register files are often implemented with the D-FF’s in the Adaptive Logic Modules (ALMs). Each ALM has two D-FFs. Therefore a 32-bit register will take up 16 ALMs. Alternatively one could also use the static memory blocks for this purpose.

Register File Internals

- For read operations, functionally the regfile is equivalent to a 2-D array of flipflops with tristate outputs on each
  - MUX, but distributed
  - Unary control

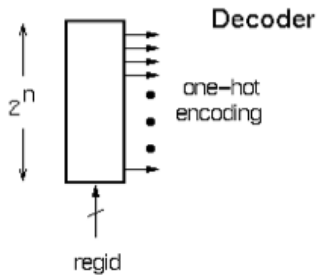


- Cell with added write logic:

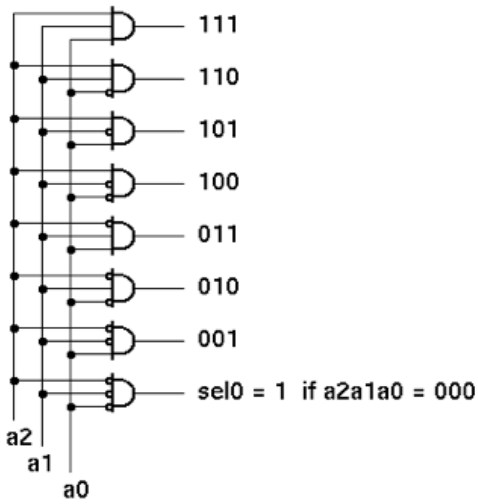


The circuit of a register file is simple – it consists of arrays of D-FFs, which can be disable (and output becomes high impedance). The register select signals sel_reg0, sel_reg1 etc. enable the correct register to put the data on the data line (called bit line here). The read/write control signal WE is used to determine if you are reading or writing to the register.

Regid (address) Decoding



regid	sel_reg1 sel_reg0
000	00000001
001	00000010
010	00000100
011	00001000
100	00010000
101	00100000
110	01000000
111	10000000

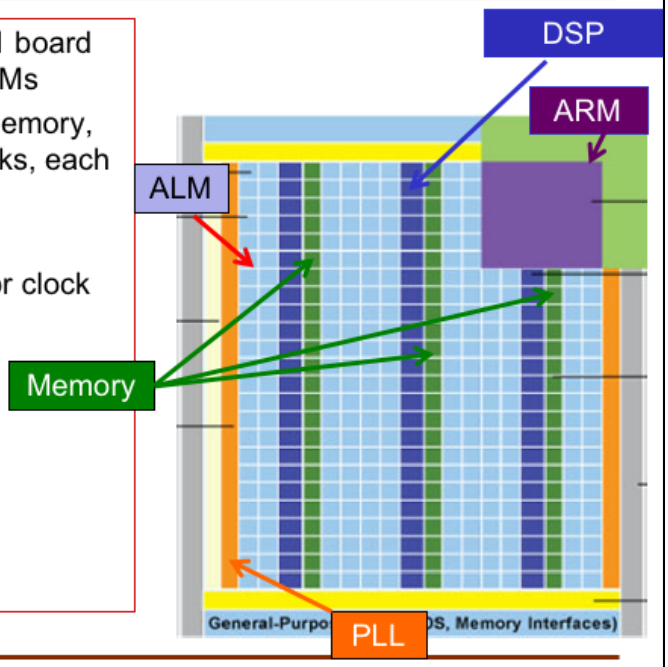


- The function of the address decoder is to generate a **one-hot code** word from the address.
  - Binary -> unary
  - Simplified DEMUX
- The output is use for row selection.
- Many different circuits exist for this function. A simple one is shown.

The register identification (regid) determines which register you are trying to access. This is achieved through a standard decoder, which generate a one-hot code word to select the appropriate register to access.

Cyclone V FPGA resources

- The Cyclone V used in the DE1 board (5CSEMA5F31C6) has 31k ALMs
- It also contains 4.45 Mbits of memory, organised as 397 memory blocks, each with 10 kbits of storage
- It has 87 DSP Blocks (later)
- It has 16 phase-locked loops for clock generation (later)



Now let us turn to the Cyclone V FPGA. The FPGA has many different type of resources in additional to Adaptive Logic Modules (ALMs). These are: memory blocks, Digital Signal Processing (DSP) units, phase-locked loops and input/output pads. In addition, there is a dual-core ARM processor and its associated bus interface circuit (shown in light green).

Here we focus on memory. In the C5-SE-A5 series, which is the one we use in the DE1 board, there are near 400 separate memory blocks, each with 10k bits of storage. Together with the ALMs, there is 4.45 Mbits of flexible memory storage available to the designer.

Cyclone V Embedded Memory

- Each 10kbit memory block (M10K) can be configured with different data width from 1 bit to 40 bit wide
- It also has multiple operating modes (which is user configurable), of which we will focus on the following only: single-port, shift-register, ROM, FIFO

- Single-port
- Simple dual-port
- True dual-port
- Shift-register
- ROM
- FIFO

Memory Block	Depth (bits)	Programmable Width
MLAB	32	x16, x18, or x20
M10K	256	x40 or x32
	512	x20 or x16
	1K	x10 or x8
	2K	x5 or x4
	4K	x2
	8K	x1

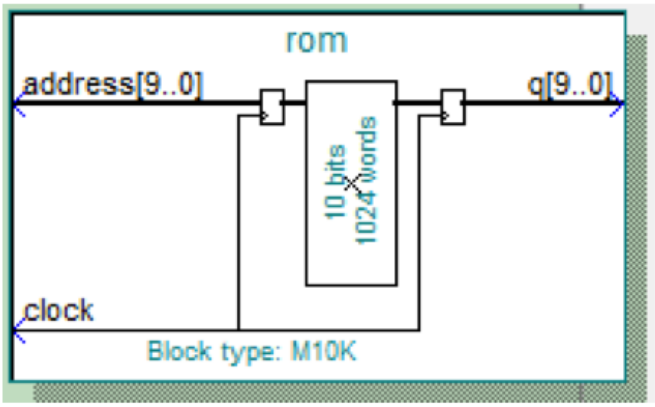
Each of these blocks (known as M10K) can be configured with different depth and data width as shown in the able above.

Even more importantly, the can also be configured to act as conventional single-port memory, or simple dual-port with one port for read and one port for write.

Further, they can be made to be true dual-port, both ports being read/write ports, or as a shift register, a ROM or a first-in-first-out buffer (FIFO).

Intialization of ROM Contents (1k x 8)

- Create ROM and initialize its content in a .mif file:




```
-- ROM Initialization file
WIDTH = 10;
DEPTH = 1024;
ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;
CONTENT
BEGIN
   0 :  200;
   1 :  203;
   2 :  206;
   3 :  209;
   4 :  20C;
   5 :  20F;
   6 :  212;
   7 :  215;
   8 :  219;
   9 :  21C;
   A :  21F;
```

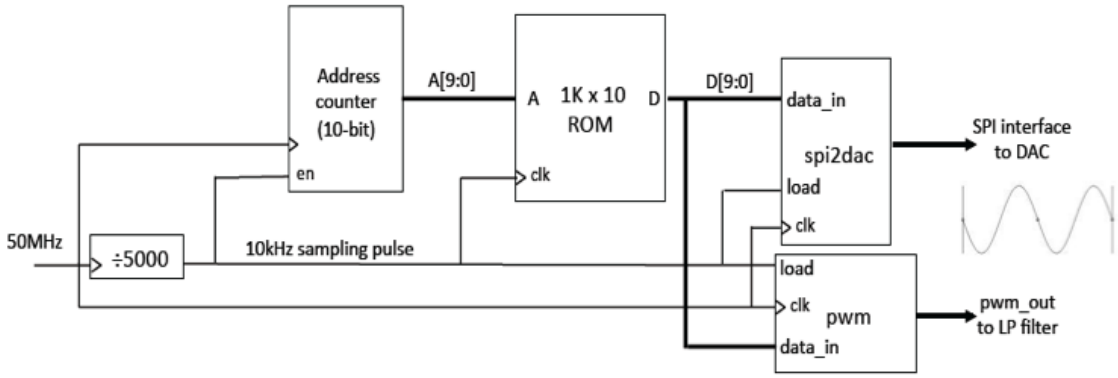

As you have seen in the VERI experiment, if the memory block is a ROM (or even as a RAM), its content can be configured via a memory initialization file .mif. The format of the file is shown here. Typing the contents of a 1024 ROM module by hand is silly and impractical. I wrote two versions of a simple programme to generate this .mif file, one in Matlab and one in Python. Below is the code for the Matlab version.

The ROM is produced using the IP Catalog tool. Here is a 1024 x 10 bit ROM generated with all input and output registered and synchronised with the clock signal.

```
% Purpose:  MATLAB script to produce contents of a ROM that stores
%           one cycle of sinewave
% Inputs:   None
% Outputs:  rom_data.mif file
% Author:   Peter Cheung
% Version:  1.0
% Date:         20 Nov 2011

DEPTH = 1024;        % Size of ROM
WIDTH = 10;          % Size of data in bits
OUTMAX = 2^WIDTH - 1;    % Amplitude of sinewave

filename = 'rom_data.mif';
fid = fopen(filename,'w');

fprintf(fid,'-- ROM Initialization file\n');
fprintf(fid,'WIDTH = %d;\n',WIDTH);
fprintf(fid,'DEPTH = %d;\n',DEPTH);
fprintf(fid,'ADDRESS_RADIX = HEX;\n');
fprintf(fid,'DATA_RADIX = HEX;\n');
fprintf(fid,'CONTENT\nBEGIN\n');

for address = 0:1023
    angle = (address*2*pi)/DEPTH;
    sine_value = sin(angle);
    data = (sine_value*0.5*OUTMAX) + OUTMAX*0.5;

    fprintf(fid,'%4X : %4X;\n',address,int16(data));
end

fprintf(fid,'END\n');
fclose(fid);
disp('Finished');
```

Sinewave Generation

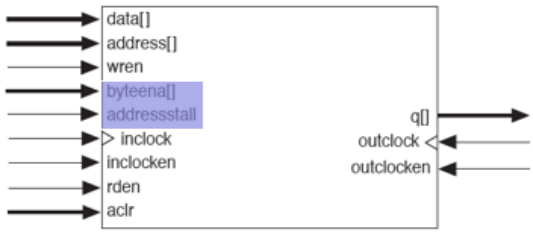
- Generate any waveform or function $y = F(x)$ using table lookup
- Phase counter increment phase whenever *step* goes high
- ROM stores one cycle of sinewave to produce F(x)
- Digital-to-Analogue convert and the PWM DAC generate the analogue outputs on L & R channels



In the experiment, you have already implemented a sine wave generator using the ROM to store one cycle of a sine wave. The counter is used to advance the phase of the sine wave, which is specified as the address X of the ROM. The content of the ROM, y= F(x) is the content of the ROM and is the generated wave form. Instead of storing a sine wave, you can easily store any other signal (such as a voice or music segment).

In order implement a variable frequency sinewave, you could modify the address counter so that it is goes up not only by 1 count for each clock cycle, but by N. For example if N is 2, then the address counter will skip every other sample in the ROM and therefore the generated sinewave will be at twice the signal frequency.

Single-port Memory (M10K as RAM)



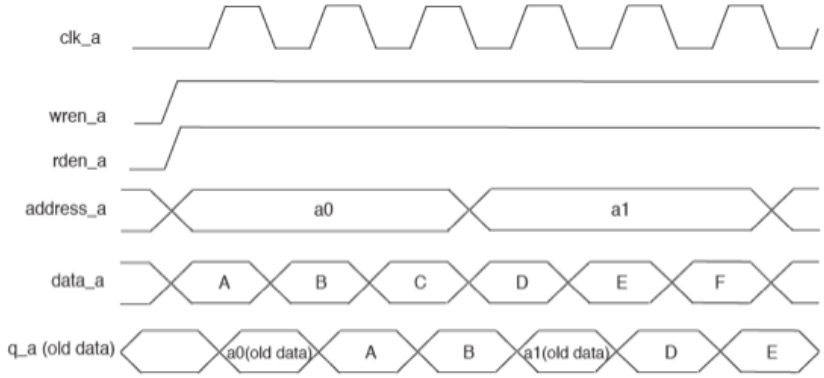
Signal name	meaning
data[ ]	Write data port
address[ ]	Read/write address port
q[ ]	Read data port
wren	Write enable
rden	Read enable
aclr	Asynchronous clear
inclock	Clock signal to control writing
outclock	Clock signal to control reading

Here is a generated single-port memory with ALL possible signals included. The meaning of all the signals are self explanatory.

Single-port Memory Timing

During a write operation, the behavior of the RAM outputs is configurable. If you activate `rden` during a write operation, the RAM outputs show either the new data being written or the old data at that address. If you perform a write operation with `rden` deactivated, the RAM outputs retain the values they held during the most recent active `rden` signal.

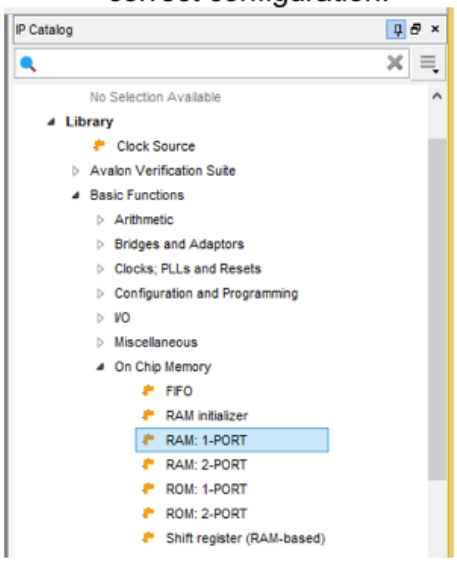
To choose the desired behavior, set the **Read-During-Write** option to either **New Data** or **Old Data** in the RAM MegaWizard Plug-In Manager in the Quartus II software.

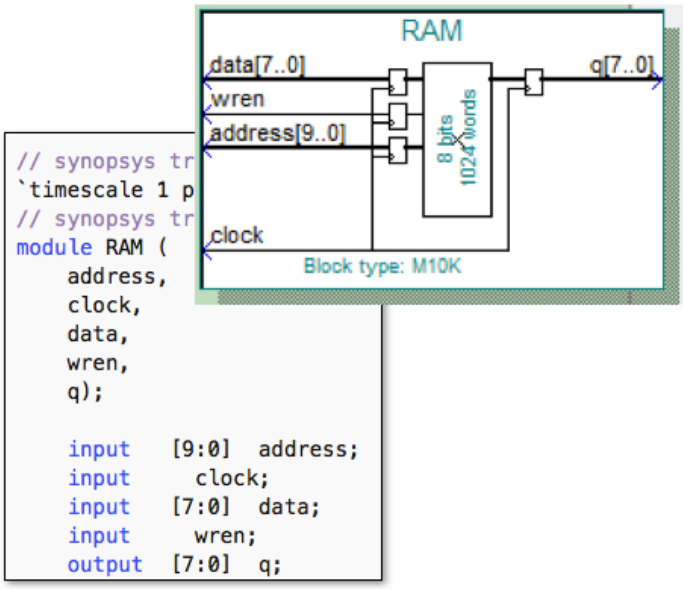


Here is the timing of the RAM configured as a single port. Since we have separate data input port (data_a) and data output port (q_a), it is important to understand what data you read back (old or new) from a given address during a write cycle.

How to use M10K memory block? (1k x 8)

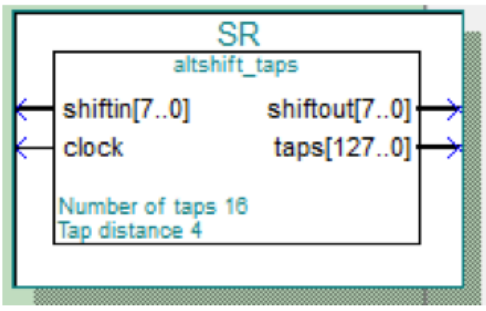
- Use IP Catalog manager tool in Quartus to produce memory of the correct configuration:



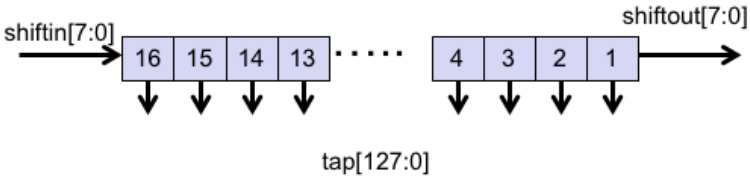


Here is an example of using the MegaWizard manager tool in Quartus. We are producing a 1-port RAM with 1024 x 8, all signals are clocked. The generator produces a sample header file (a template) which defines the interface signal to the generated block. Remember you must tick the Verilog HDL radio button.

M10K Memory as Shift Register (8-bit 16 stages)



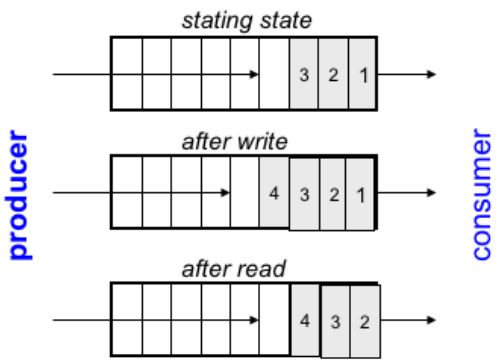

```
module SR (
    clock,
    shiftin,
    shiftout,
    taps);

    input     clock;
    input   [7:0]  shiftin;
    output  [7:0]  shiftout;
    output  [127:0]  taps;
```



You can also configure the M9K memory block as a shift register. Here is an 8-bit 16 stage SR. In addition, it provides “tap” outputs for every stage, i.e. 16 x 8 = 128 output signals. This is very useful to implement FIR filter or perform time domain convolution.

First-in-first-out (FIFO) Memory

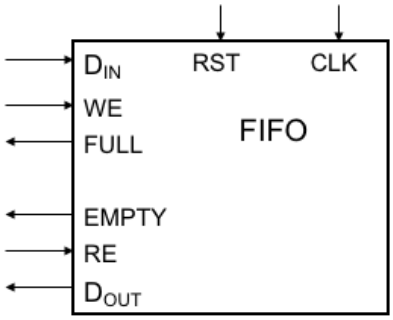
- Used to implement *queues*.
- These find common use in computers and communication circuits.
- Generally, used for rate matching data producer and consumer:



- Producer can perform many writes without consumer performing any reads (or vice versa). However, because of finite buffer size, on average, need equal number of reads and writes.
- Typical uses:
  - interfacing I/O devices. Example network interface. Data bursts from network, then processor bursts to memory buffer (or reads one word at a time from interface). Operations not synchronized.
  - Example: Audio output. Processor produces output samples in bursts (during process swap-in time). Audio DAC clocks it out at constant sample rate.

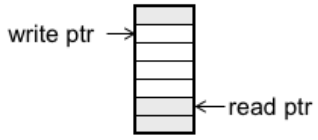
In the Part IV of the VERI experiment, you will be using a FIFO to implement an echo synthesizer. The action of a FIFO is shown in the diagram above.

FIFO Interfaces

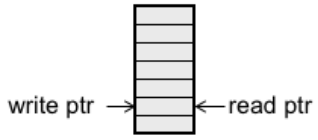


- After write or read operation, FULL and EMPTY indicate status of buffer.
- Used by external logic to control own reading from or writing to the buffer.
- FIFO resets to EMPTY state.

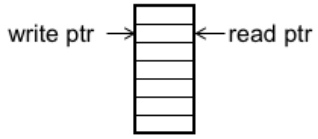
- Address pointers are used internally to keep next write position and next read position into a dual-port memory.



- If pointers equal after write ⇒ FULL:

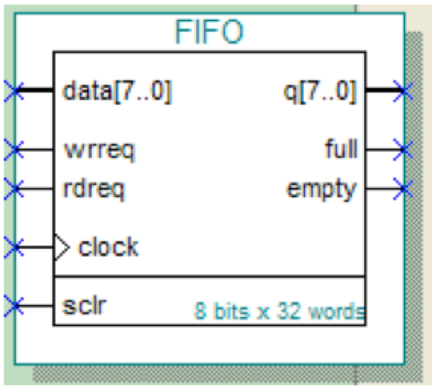


- If pointers equal after read ⇒ EMPTY:



Here is a generic block diagram of a FIFO with its typical interface signals. FIFO is a form of queue. Internally there typically two counters, one keeping track of the read address (or read pointer) and another counter keeping track of the write address (write pointer). There needs to be status signals such as FULL, which is asserted if the FIFO is completely filled and writing any more words to it will destroy stored data, or EMPTY, which signifies that there are no data left to read.

M10K Memory as FIFO (8-bit x 32 word)




```
module FIFO (
    clock,
    data,
    rdreq,
    sclr,
    wrreq,
    empty,
    full,
    q);

    input      clock;
    input    [7:0]  data;
    input      rdreq;
    input      sclr;
    input      wrreq;
    output     empty;
    output     full;
    output   [7:0]  q;

endmodule
```


FIFO can be generated using the IP Catalog manager tool. Here is an example of a 32 word x 8 bit FIFO.