Imperial College London

Lecture 15

Memories

Peter Cheung
Imperial College London

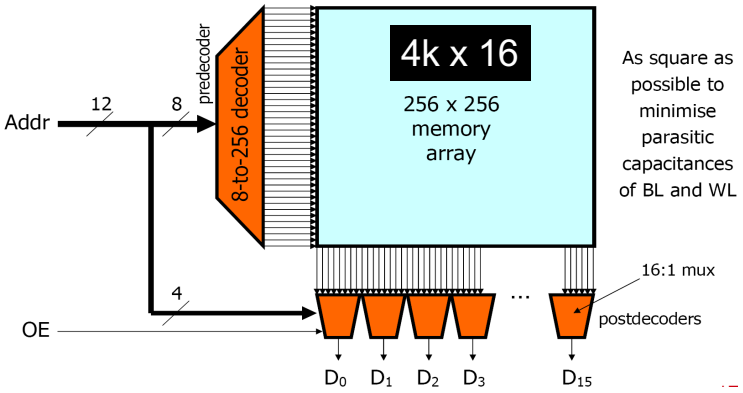
URL: www.ee.imperial.ac.uk/pcheung/teaching/E2_CAS/
E-mail: p.cheung@imperial.ac.uk

Lecture Objectives

- Explain the sequence of events in reading from and writing to a static RAM
- Explain the structure and input/output signals of a static RAM
- How to design an address decoder
- Investigate the timing diagrams for a microprocessor when reading from or writing to memory
- Explain how the embedded memory in an FPGA can be used to implement memory blocks in a digital design

Memory interfacing is an essential topic for digital system design. In fact the among silicon area devoted to memory in a typical digital embedded system or a computer system is substantial. For example, in a mobile phone, the number of transistors devoted to memory is many times more than those used for computation. For the second year course, I will only focus on interfacing to static memory, known as RAM (Random Access Memory) or ROM (Read-Only Memory). There are other types of memory such as dynamic memory (DRAM), Synchronous DRAM (SDRAM) and flash memory (Flash RAM) which will not be covered on this course.

Simplified RAM Organization



A typical memory has:

- N-bit address bus
  - This defines 2^N memory locations
- M-bit data bus
  - This defines the size of data word
- Control signals to define read/write cycles,
  - Asynchronous memory – no clock in control
  - Synchronous memory – every action synchronised to a clock

A typical 8-bit microprocessor typically has

- A 16-bit address bus, A15:0
  - Can have up to 2^{16}=65536 memory locations
- An 8-bit data bus, D7:0 - Each data word in memory has 2^8 = 256 possible values
- In the RAM shown above uses 12-bit address and 16-bit data, i.e. 4096 locations of 16-bits each
- These are arranged as 256 x 256 rows of memory cells. 4096 = 256 rows x 16 columns as shown
- The address bus is therefore split into two components: 8-bit to specify which row, and 4-bit to select the correct column.

This slide shows a typical organisation inside a RAM chip. Memory cells are usually organised in the form of a 2-D array of RAM cells. These are accessed first in a row, then in a column. The address bus is divided into two components, the row address (8-bit in the example here) and the column address (4-bit in this example). There is a decode to translate the 8-bit row address into one-hot outputs in order to specify which row is being accessed. Only ONE ROW will be enable at any one time (hence one-hot).

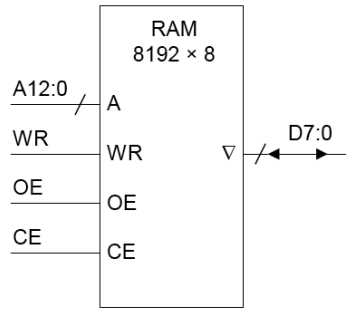
The second part of the address (normally the less significant bits) is used as select signal into the output mux. This is because when memory is accessed, they are normally read or written in a sequence. Using LSB for column decoding means that one stays on the same row of memory as much as possible. Staying in the same row uses significantly lower energy than switching between rows in memory accesses.

In the example here, the 4-bit column address is used to select from a 16-to-1 mux to provide the correct location in memory to access. There are 16 identical blocks, each providing one-bit of the data output.

The output enable signal OE allows the selected data value be driven on the data bus.

RAM: Read/Write Memory

8k × 8 Static RAM



CE	OE	WR	D0:7	Action
0	?	?	Hi Z	Disabled
1	0	0	Hi Z	Idle
1	1	0	Out	Read
1	?	1	In	Write

Hi Z = High impedance

Static RAM:	Data stored in bistable latches
Dynamic RAM:	Data stored in charged capacitors: retained for only 2ms. Less circuitry ⇒ denser ⇒ cheaper.

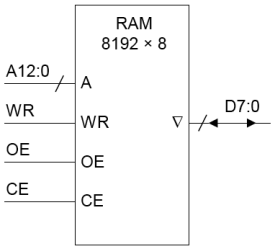
∇	Tri-state output: Low, High or Off (High Impedance). Allows outputs from several chips to be connected; Designer must ensure only one is enabled at a time.
CE	Chip Enable: disabling chip cuts power by 80%.
OE	Output Enable: Turns the tri-state outputs on/off.
A12:0	Address: selects one of the 2^{13} 8-bit locations.
WR	Write: stores new data in selected location
D7:0	Data in for write cycles or out for read cycles.

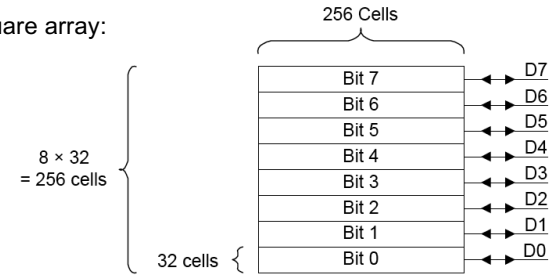
Here is a 8K x 8 static RAM chip and its associated digital signals. The 13-bit address bus A12:0, the 8-bit data bus D7:0 are mandatory. There are three more control signals: Output Enable OE which we have seen before, Chip Enable CE which is used to address or select this particular memory chip (hence the name), and finally the WRITE ENABLE signal WE, which, when set high, indicates that you are writing to the RAM chip, and is normally low (i.e. reading).

Note that the data bus has an inverted triangle sign, indicating that this is a **tri-state bus**. This means that the pin could be an input pin, output pin, or an open-circuit pin (i.e. not connected to anything – we call the signal *floating*). The truth table shown here specifies the behaviour of the data bus in one of the three possible states.

8k x 8 Static RAM

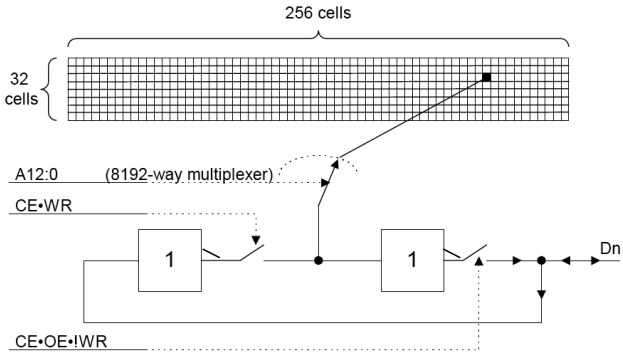
- The 64k memory cells are arranged in a square array:





- For each output bit, an 8192-way multiplexer selects one of the cells. The control signals, **OE**, **CE** and **WR** determine how it connects to the output pin via buffers:

- Occasionally DIN and DOUT are separate but ⇒ more pins



PYKC 3 Dec 2024 EE2 Circuits & Systems Lecture 15 Slide 5

For a 8k x 8 RAM, there are 8 data bits, and therefore 8 separate 1-bit arrays. Let us assume that each data bit array is organised as a 256 rows x 32 column (=8192) of memory cells. Eight such array are placed next to each other to form the 8 data bits required. This makes the memory chip roughly square (which is generally desirable).

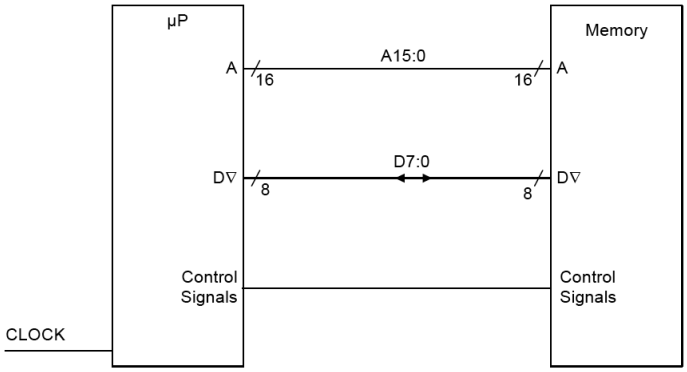
You can think of the row decoder and the column selector driven by the 13-bit address as a 8192 way multiplexer, selecting one of 8192 cells organised as 256 x 32, to be accessed.

The simplified circuit of each memory cell shown here consists of two inverters and two switches is a schematic of the read-write circuit. When reading from the cell, A12:0 select one of 8192 cells to route its signal via the right inverter to Dn. Now Dn is an output pin. This only happens if CE*OE* !WR = 1 (i.e. asserting CE and OE, but not asserting WR).

When writing to the memory cell, the right switch is open, Dn is an input pin driving the left hand inverter and the output switch from that inverter is closed because both CE and WR are asserted.

Some memory chips have separate Din and Dout pins, but that's expensive on pins and is not particularly common nowadays.

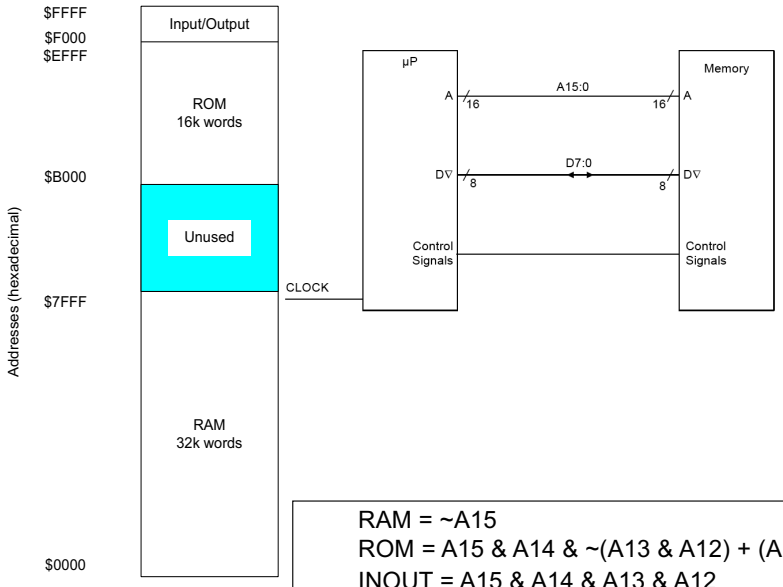
Microprocessor ⟷ Memory Interface



- During each memory cycle:
- A15:0 selects one of 2^{16} possible memory locations
- D7:0 transfer one word (8 bits) of information either to the memory (write) or to the microprocessor (read).
- D7:0 connections to the microprocessor are tri-state (∇): they can be:
  - "logic 0", "logic 1" or "high impedance" (inputs)
- The control signals tell the memory what to do and when to do it.

Here is a slide showing a generic interfacing between a microprocessor and a memory sub-system. We assume that we use a 16-bit address bus and an 8-bit data bus. The control signals go between the two to control the transfer of information, and is in general governed by the microprocessor which acts as the "**master**".

Microprocessor Memory Map



We can tell which region of memory an address is in by inspecting the top few bits:

	A15:12	
F:	1111	Input/Output
E:	1110	ROM
D:	1101	ROM
C:	1100	ROM
B:	1011	ROM
A:	1010	
9:	1001	
8:	1000	
7:	0111	RAM
6:	0110	RAM
5:	0101	RAM
4:	0100	RAM
3:	0011	RAM
2:	0010	RAM
1:	0001	RAM
0:	0000	RAM

RAM = ~A15
ROM = A15 & A14 & ~(A13 & A12) + (A15 & ~A14 & A13 & A12)
INOUT = A15 & A14 & A13 & A12

While we show memory as a block, in a real system, the memory address space is divided into many different partitions. Here we use '$' (instead of 16'hxxxx) to indicate that the addresses are hexadecimal numbers. The left hand diagram shows the memory being partitioned into 32k of RAM, 16k of ROM and 4k space for input/output devices.

A design needs to take **the upper bits of the address bus** and decode these bits into **enable signals** for the three different partitions. In this case, we can see that we only need to decode A15:12 according to the Boolean equations shown here. What about A11:0? These are the address bits used inside the RAM, ROM and input/output modules to select particular locations.

Memory Chip Selection

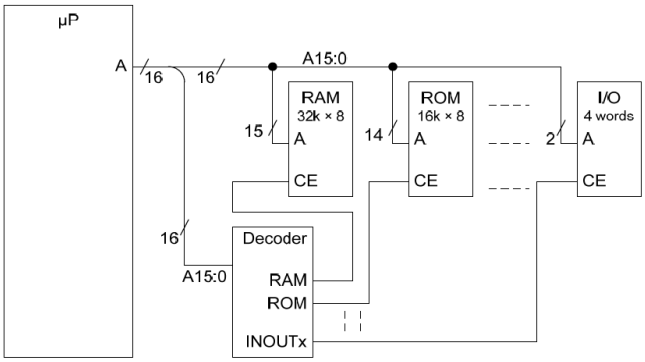
- Each memory circuit has a "chip enable" input (CE)
- The "Decoder" uses the top few address bits to decide which memory circuit should be enabled. Each one is enabled only for the correct address range:

  RAM = ~A15

  ROM = A15 & A14 & ~(A13 & A12) + (A15 & ~A14 & A13 & A12)

  INOUTx = **A15 & A14 & A13 & A12** & ~A11 & A10 & ~A9 & A8 & ~A7 & A6 & A5 & A4 &~A3 & A2
- INOUTx responds to addresses: $F574 to $F577
  other I/O circuits will have different addresses
- Low *n* address bits select one of 2^n locations within each memory circuit (value of n depends on memory size)

Addr Range	Usage
$F578 - $FFFF	Not used
$F574 - $F577	**INOUTx**
$F000 - $F573	Not used
$B000 - $EFFF	16k ROM
$8000 - $AFFF	Not used
$0000 - $7FFF	32k RAM



Selecting which memory sub-system and therefore which memory chip to enable is the job of the address decoder circuit. This circuit takes the upper bits of the address bus, and produce enable signals for RAM, ROM and INOUTx for a particular I/O device.

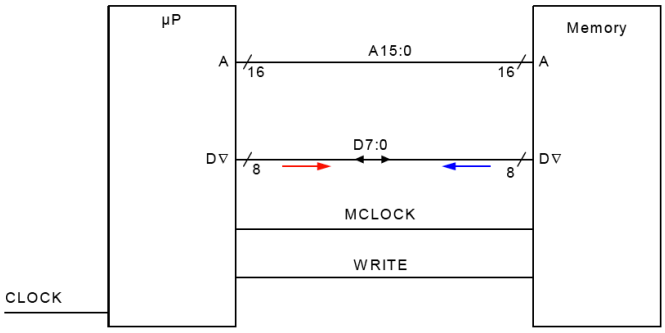
In the previous slide, we showed that the input/output occupies 4k of memory space. This is uncommon. Typically an I/O device may take up, say, 4 memory locations.

In this example, INOUTx occupies only the address space $F574 - $F577 (hexadecimal format) , i.e. 4 locations. Therefore we need to decode lots of address signals: A15:2.

Can you work out the Boolean equations for the address decoder shown here?

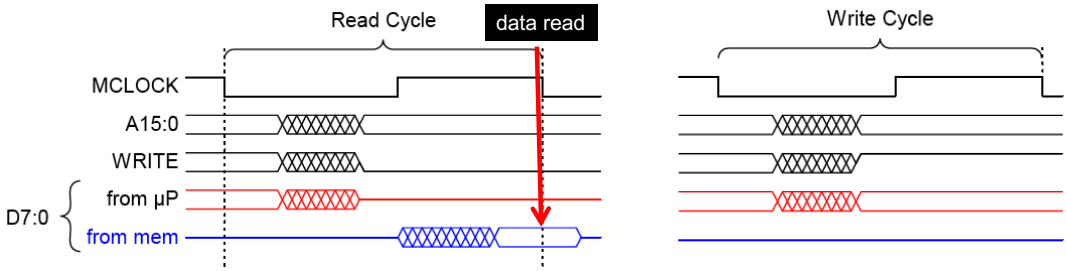
The ROM CE signal is another challenge. The ROM is enable if the address A15:A12 falls between the range 4'b1011 and 4'b1110. You should prove for yourself that the Boolean equation to decode the address for the ROM is as shown here.

Memory Interface Control Signals



Control signals vary between µProcessors but all have:

- A clock signal to control the timing (can be the same as the system CLOCK)
- A signal to say whether the microprocessor wants to read from memory or to write to memory
  - Must make sure that D7:0 is only driven at one end



D7:0 from memory only allowed when MCLOCK & ~WRITE true

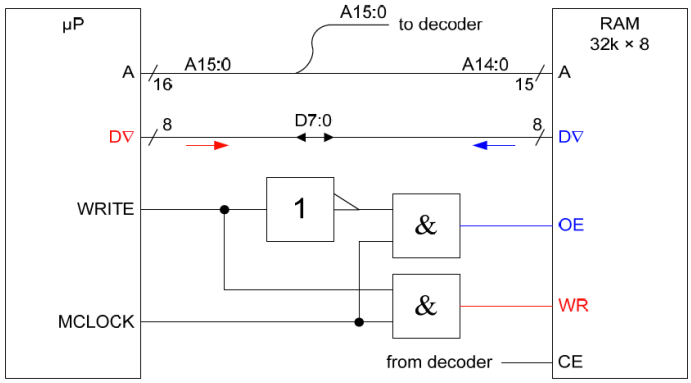
In addition to the address decoder circuit, we need to provide the control signals from the microprocessor to the memory chips. Here we assume there exists at least two control signals from the microprocessor: MCLOCK which is memory clock signal (which may be different from the system clock signal CLOCK), and a WRITE signal, which is high when writing to memory, but low otherwise.

The interaction between the microprocessor and memory can be separated into two types of transactions: a **Read Cycle** and a **Write Cycle**.

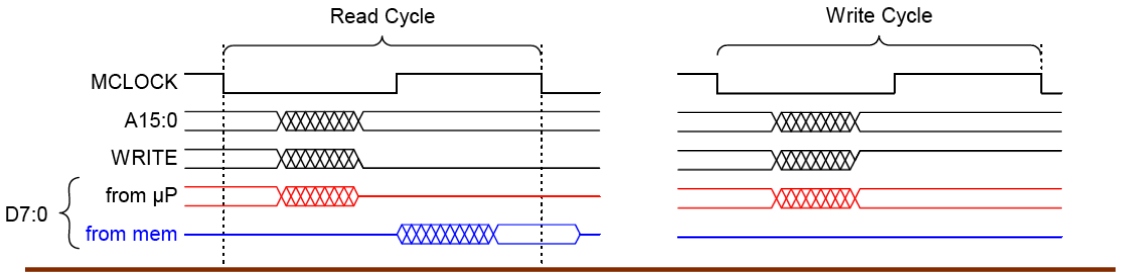
During Read Cycle, the microprocessor asserts the address A15:0 and the control signals MCLOCK and WRITE. Shortly after the beginning of the Read Cycle, the microprocessor must STOP driving the data bus D7:0, and on the second half of the cycle, we assume that memory will then provide the data for the microprocessor to read. Reading is actually performed at the end of the Read Cycle, on the falling edge of MCLOCK. Note that I use red colour to indicate the action of the microprocessor on the data bus, and blue colour for the action by the memory chip on the data bus.

During a Write Cycle, the microprocessor drives everything. Writing also occurs on the falling edge of MCLOCK in our case. (Note that other system may have a different protocol than the one shown here.)

Memory Circuit Control Signals



- **Output enable**: OE = MCLOCK & ~WRITE turns on the D7:0 output from the memory
- **Write enable**: WE = MCLOCK & WRITE writes new information into the selected memory location with data coming from microprocessor
- **Chip enable**: comes from the decoder and makes sure the memory only responds to the correct addresses

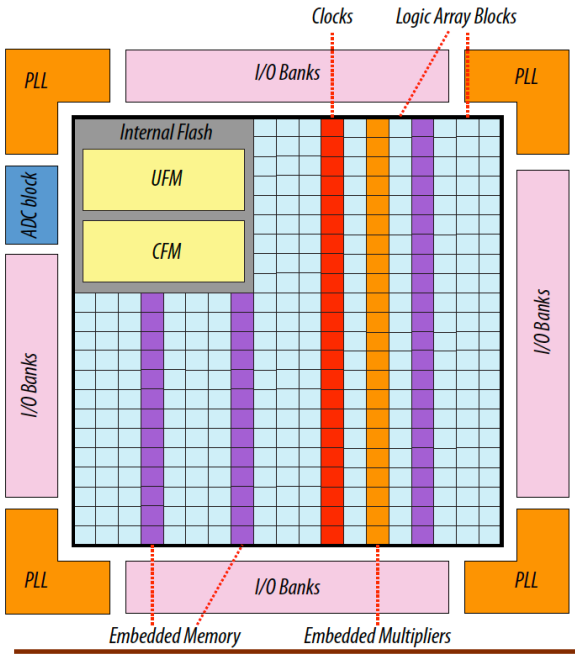


PYKC 3 Dec 2024 EE2 Circuits & Systems Lecture 15 Slide 10

This slide shows the control circuit used to interface the microprocessor to the 32k x 8 RAM chip.

Chip Enable (CE) is driven by the output from the address decoder, which we have considered in an earlier slide. Remember the colour code I am using: RED driven by the microprocessor, BLUE driven by memory.

Max 10 FPGA – Embedded blocks



- MAX10 device: 10M50DAF484C7G
- 50,000 Logic Elements (4-LUT + FF)
- 182 M9K embedded memory blocks
- 5,888 kbits user flash memory
- 144 hard multiplier (18 x 18)
- 4 PLL (for clock generation

The MAX10 FPGA device we use in DE10-Lite board has nearly 50,000 Logic Elements, each consisting of a 4-input Lookup Table (4-LUT) and a D-FF.

There are also blocks of memory (shown in purple) and multipliers (in brown). Note that the internal structure follows the traditional array style with rows and columns. Each column has the SAME type of circuit (i.e. all LEs or all multipliers). For our MAX 10 chip, there are 182 9-bit memory blocks, and 144 18-bit x 18-bit hardware multipliers.

On the edge are lots of programmable I/Os. These can be configured for different logic standards, as input or output, have different current drive strengths and slew rates.

There are four phase-locked loops (PLLs) used for generating internal clock signals.

There is a analogue-to-digital converter block for interfacing to analogue signals. However, we are not using this in our Lab experiments.

The internal flash memory blocks are used to store program codes for soft 32-bit processors called Nios II. Unlike some other FPGAs with inbuilt ARM process, which is a hard block, MAX 10's Nios II processor is implemented with LEs. If it is not needed, the configurable logic can be used for other purposes.

MAX 10 Embedded Memory

- Each 9kbit memory block (M9K) can be configured with different data width from 1 bit to 36 bit wide
- It also has multiple operating modes (which is user configurable), of which we will focus on the following only: 1-port ROM, FIFO, 2-port RAM

Configuration	M9K Block
Depth × width	8192 × 1
	4096 × 2
	2048 × 4
	1024 × 8
	1024 × 9
	512 × 16
	512 × 18
	256 × 32
	256 × 36

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

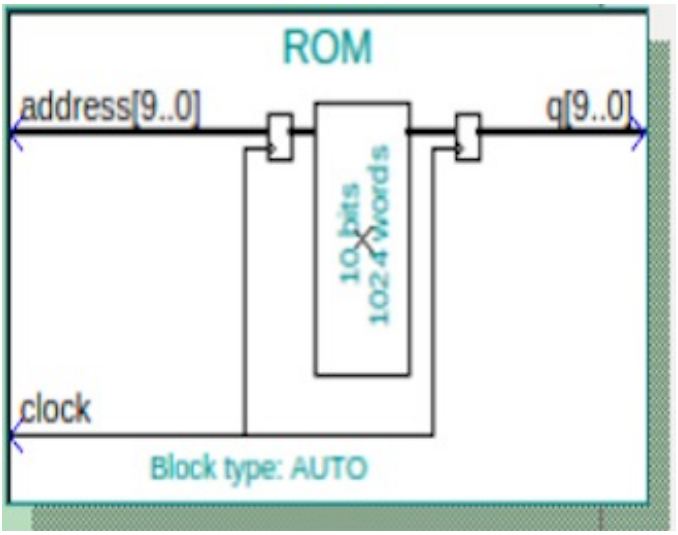
Each of these blocks (known as M9K) can be configured with different depth and data width as shown in the able above.

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. They can also be configure as *true dual-port* RAM, where both ports can be read or write ports.

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;
```

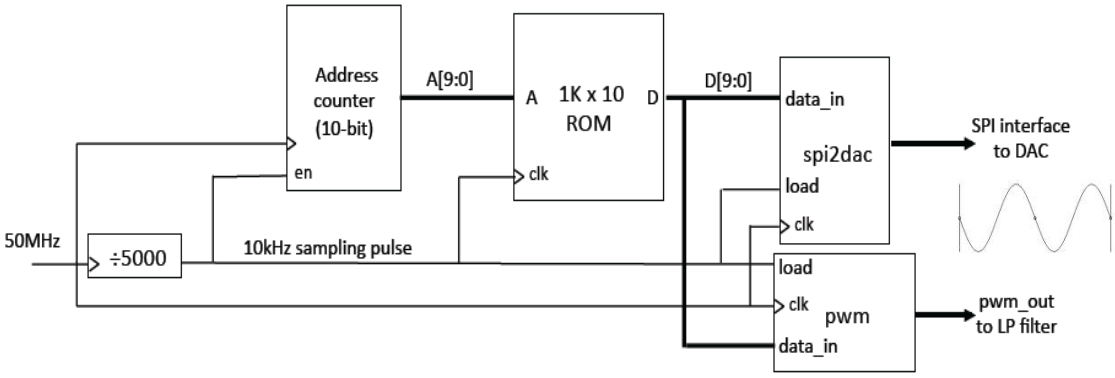

As you have seen in the VERI experiment, if the memory block is a ROM (or even as a RAM), its content can be configured via a memory initialization file .mif. The format of the file is shown here. Typing the contents of a 1024 ROM module by hand is silly and impractical. I wrote two versions of a simple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

```
% 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

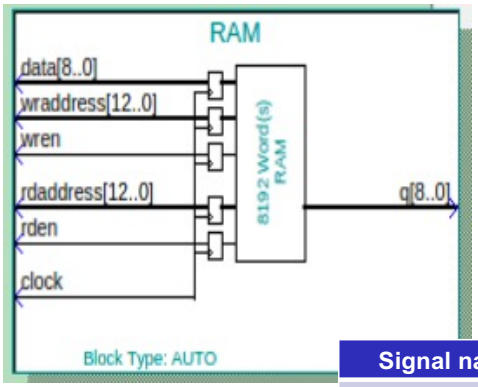
- Generate any waveform or function $y = F(x)$ using table lookup
- Phase counter increment phase whenever *step* goes high
- ROM stores one cycle of sinewave to produce F(x)
- Digital-to-Analogue convert and the PWM DAC generate the analogue outputs on L & R channels



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.

Dual-port RAM (8k x 9)



- Limited data width (1 to **9**, 16, 18, 32, 36 …)
- Depth up to 65,536 (4-bit to 16-bit address)
- **Simple dual-port: 1 read port & 1 write port**
- True dual-port: both ports can read/write

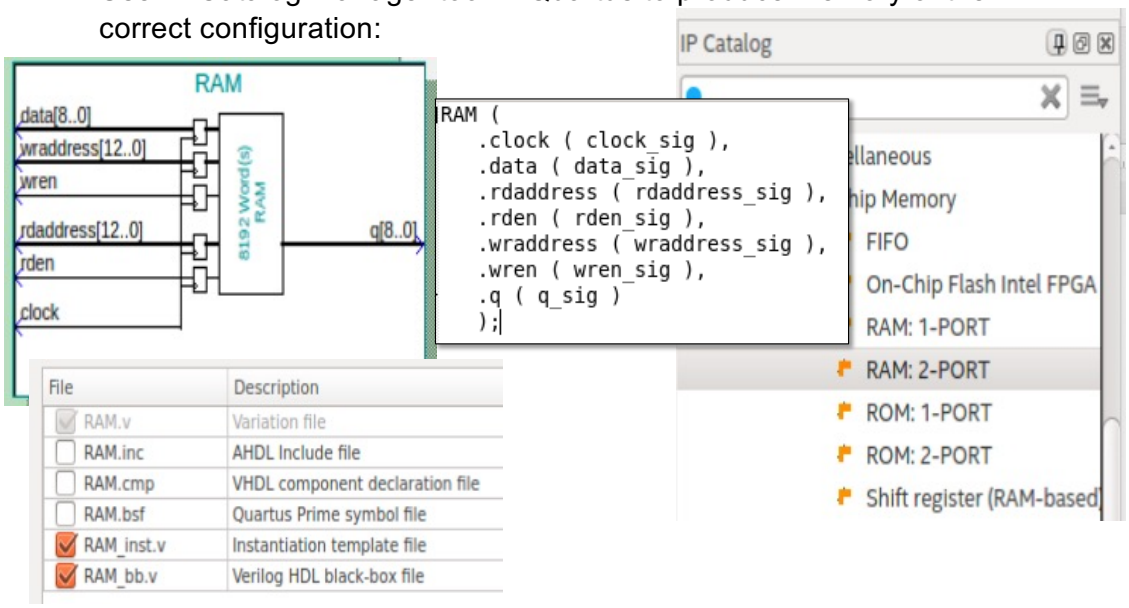
Signal name	meaning
data[ ]	Write data port
address[ ]	Read/write address port
q[ ]	Read data port
wren	Write enable
rden	Read enable
clock	Clock signal to control both read & write

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

A simple dual-port memory has one port for read an done for write. There are however restrictions on the number of bits in each data word, depending on the configuration of the memory. For example, if one want to use block memory as simple dual-port RAM, the data width are limited to those show in the slide. There is no option use a 10-bit memory – we have to design using only 9-bit data!

How to use M9K memory block? (8k x 9)

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



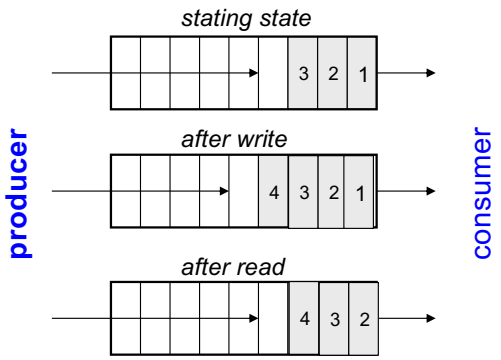
Here is an example of using the MegaWizard manager tool in Quartus. We are producing a 1-port RAM with 8k x 9 bits, all signals are clocked. The output q[8:0] is however NOT clocked in this case.

The generator produces a instantion template file which defines the interface signal to the generated block as shown. Remember you must tick the Verilog HDL radio button for this to be produced. You may copy and paste this template to your top-level design to instantiate this RAM block.

File	Description
RAM.v	Variation file
RAM.inc	AHDL Include file
RAM.cmp	VHDL component declaration file
RAM.bsf	Quartus Prime symbol file
RAM_inst.v	Instantiation template file
RAM_bb.v	Verilog HDL black-box file

First-in-first-out (FIFO) Memory

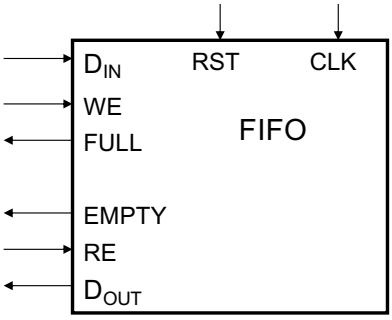
- Used to implement *queues*.
- These find common use in computers and communication circuits.
- Generally, used for rate matching data producer and consumer:



- 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.

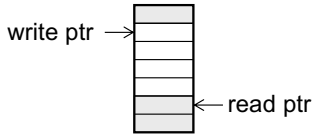
In Lab 6 of the practical for this module, you would have used a FIFO to implement an echo synthesizer. The action of a FIFO is shown in the diagram above.

FIFO Interfaces

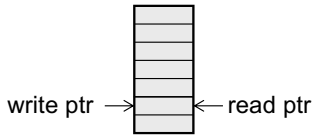


- After write or read operation, FULL and EMPTY indicate status of buffer.
- Used by external logic to control own reading from or writing to the buffer.
- FIFO resets to EMPTY state.

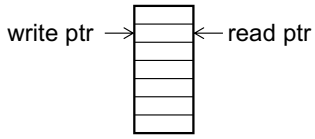
- Address pointers are used internally to keep next write position and next read position into a dual-port memory.



- If pointers equal after write ⇒ FULL:

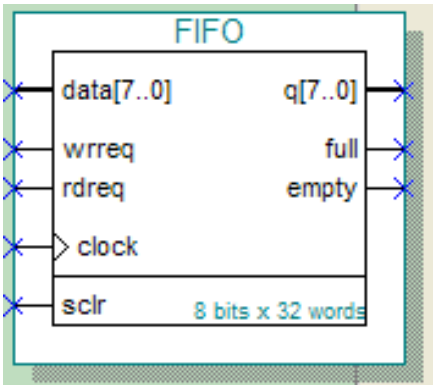


- If pointers equal after read ⇒ EMPTY:



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.

M9K 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. Shown here is an example of a 32 word x 8 bit FIFO.