Home » Tutorials » DDR3 Memory

# DDR3 Memory

2024-09-23 · 28 min · Justin Rajewski | Suggest Changes

▶ Table of Contents

In this tutorial we are going to set up an interface to the DDR3 memory with the FPGA on the Alchitry Au/Au+.

## Setup

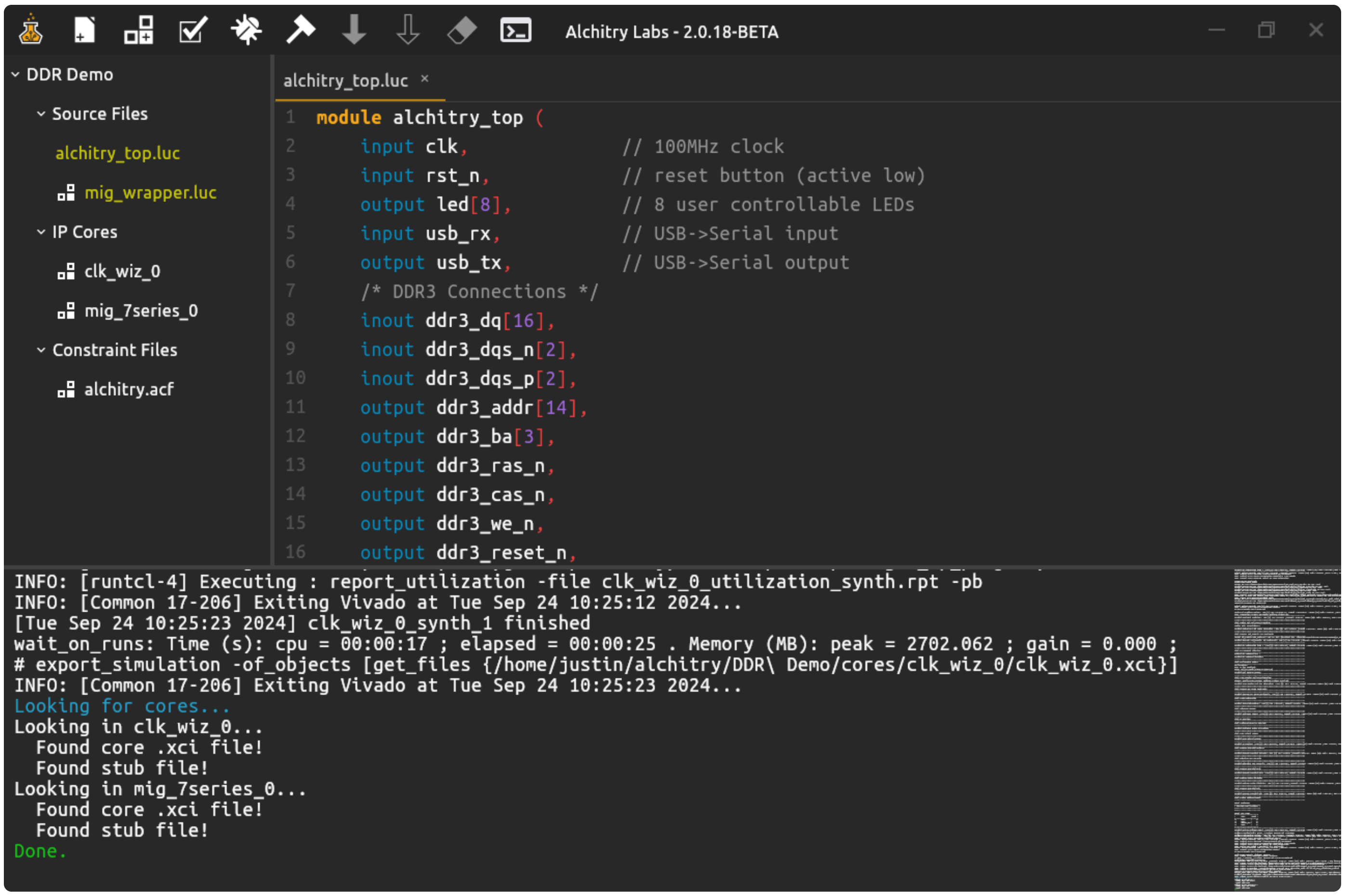
The first step is to create a project.

Open Alchitry Labs, and create a new project based on the *DDR3 Base Project*. I called mine *DDR Demo*, but feel free to name yours whatever you want.

This project temple is a bit special in that it includes some *Vivado IP Cores*. The cores themselves aren't included in the temple but scripts for their creation are. These are fired off when the project is created.

Make sure to have Vivado setup correctly before creating the project or the IP cores will fail to generate, and you'll have to delete your project and start fresh.

When the project first opens, there will be some errors reported in `alchitry_top` and the `mig_wrapper` . These will clear up once the IP cores have finished generating. This can take a few minutes.



```
module alchitry_top (
    input clk,              // 100MHz clock
    input rst_n,            // reset button (active low)
    output led[8],          // 8 user controllable LEDs
    input usb_rx,           // USB->Serial input
    output usb_tx,          // USB->Serial output
    /* DDR3 Connections */
    inout ddr3_dq[16],
    inout ddr3_dqs_n[2],
    inout ddr3_dqs_p[2],
    output ddr3_addr[14],
    output ddr3_ba[3],
    output ddr3_ras_n,
    output ddr3_cas_n,
    output ddr3_we_n,
    output ddr3_reset_n,
```

```
INFO: [runtcl-4] Executing : report_utilization -file clk_wiz_0_utilization_synth.rpt -pb
INFO: [Common 17-206] Exiting Vivado at Tue Sep 24 10:25:12 2024...
[Tue Sep 24 10:25:23 2024] clk_wiz_0_synth_1 finished
wait_on_runs: Time (s): cpu = 00:00:17 ; elapsed = 00:00:25 . Memory (MB): peak = 2702.062 ; gain = 0.000 ;
# export_simulation -of_objects [get_files {/home/justin/alchitry/DDR\ Demo/cores/clk_wiz_0/clk_wiz_0.xci}]
INFO: [Common 17-206] Exiting Vivado at Tue Sep 24 10:25:23 2024...
Looking for cores...
Looking in clk_wiz_0...
  Found core .xci file!
  Found stub file!
Looking in mig_7series_0...
  Found core .xci file!
  Found stub file!
Done.
```

Once it finishes, you should see the `clk_wiz_0` and `mig_7series_0` cores added to the project.

## The Memory Controller

There is special hardware on the Artix 7 FPGA that is used for interfacing with the DDR chip. To efficiently use this, Xilinx provides customizable IP via their IP catalog in Vivado.

Customizing this IP requires full knowledge of the DDR3 chip being used and the board pinout. While you can find all the information you need to set this up for the Alchitry Au, we have drastically simplified it for you in Alchitry Labs.

The controller was added automatically as part of this project template, but you can add it manually to another project by clicking the three block icon and selecting *Generate MIG Core (DDR)* from the dropdown. Note that this option is missing for our project since the core has already been added.

If you look at the *IP Cores* section of the project tree, you should see the *mig_7series_0* core. You can double-click on it to open its stub file. This is an empty Verilog file that defines all the connections to the core.

## Adapting to Lucid

You can use this core directly in your design if you want, but it is generally more convenient to wrap it. The `mig_wrapper` component does just that.

```
global Memory {
    struct in {
        addr[28],
        cmd[3],
        enable,
        wr_data[128],
        wr_enable,
        wr_mask[16]
    }

    struct out {
        rd_data[128],
        rd_valid,
        rdy,
        wr_rdy
    }
}
```

This component wraps the `mig_7series_0` core and defines two global structures to make hooking up other modules to it easier.

If you look at the module declaration of the wrapper, you will see a bunch of `inout` and `output` ports that start with *ddr3_*. These are all top level signals that need to connect to the DDR3 chip.

These names are actually important since the pins they connect to are defined in the `mig_7series_0` core. You don't need to specify their pinouts in your constraints file.

## Hooking up the wrapper

The basic connections for the wrapper are already handled for you in `alchitry_top`.

```
module alchitry_top (
    input clk,              // 100MHz clock
    input rst_n,            // reset button (active low)
    output led[8],          // 8 user controllable LEDs
    input usb_rx,           // USB->Serial input
    output usb_tx,          // USB->Serial output
    /* DDR3 Connections */
    inout ddr3_dq[16],
    inout ddr3_dqs_n[2],
    inout ddr3_dqs_p[2],
    output ddr3_addr[14],
    output ddr3_ba[3],
    output ddr3_ras_n,
    output ddr3_cas_n,
    output ddr3_we_n,
    output ddr3_reset_n,
    output ddr3_ck_p,
    output ddr3_ck_n,
```

Again, it is important you don't change the names of these signals since they are used in the constraint file included with the `mig_7series_0` core. If you change the names, the tools won't know what pins they need to connect to on the FPGA.

## Setting up the clock

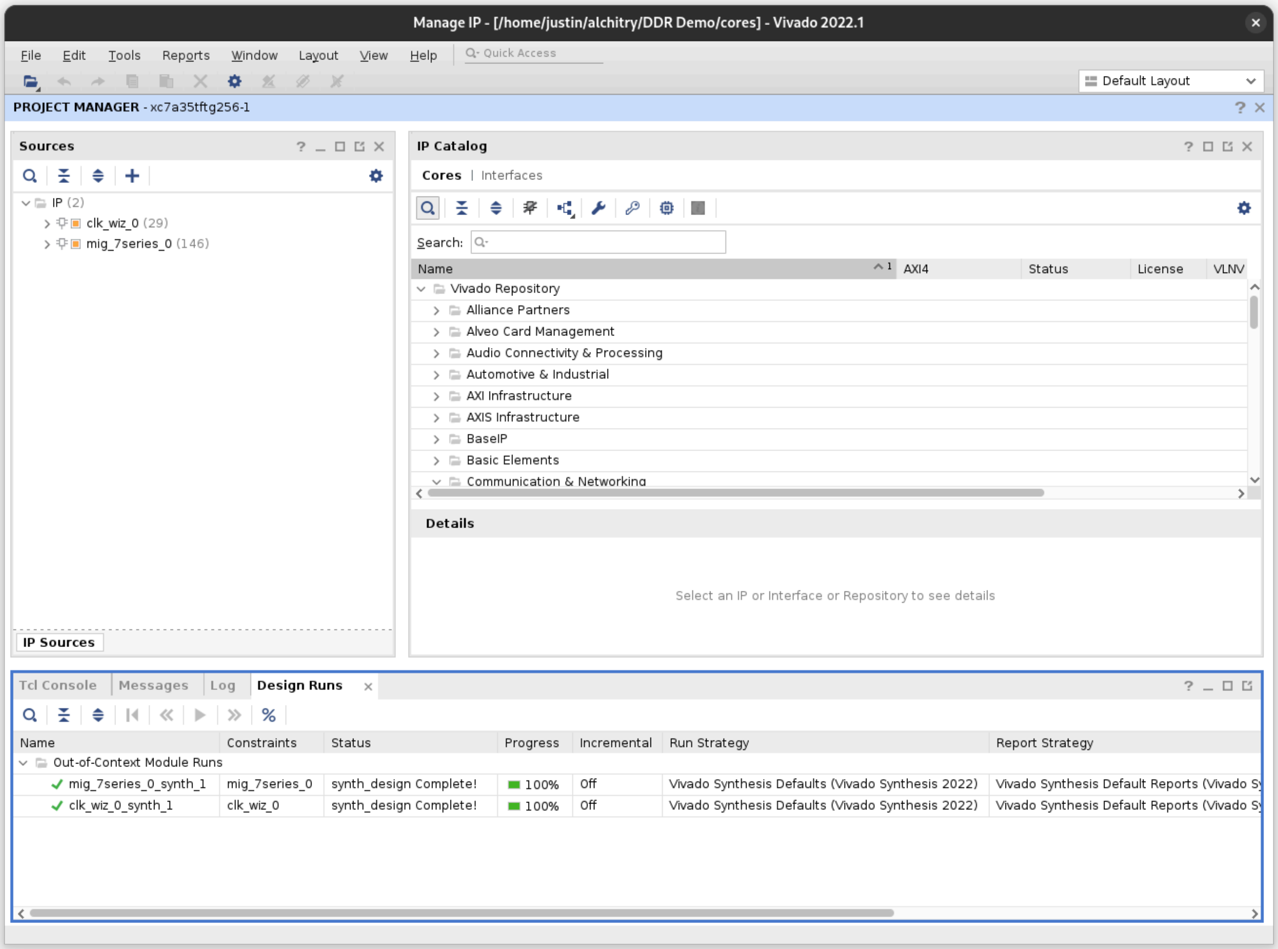
The memory interface requires both a 100MHz and a 200MHz clock. Since the Alchitry Au only has a 100MHz clock, we need to synthesize the 200MHz one.

This is the purpose of the `clk_wiz_0` core.

This core takes in a 100MHz clock and outputs 100MHz and 200MHz clocks. You may be thinking that outputting the same frequency is pointless, but the way the clock is routed in the FPGA means that the input clock can't be used anywhere else. If you need 100MHz somewhere else, you need to get it from the `clk_wiz_0` .

The FPGA can actually synthesize a lot of different frequencies. You can check out the wizard by opening the *Vivado IP Catalog*. Click the three block icon in the toolbar and select *Vivado IP Catalog* from the dropdown.

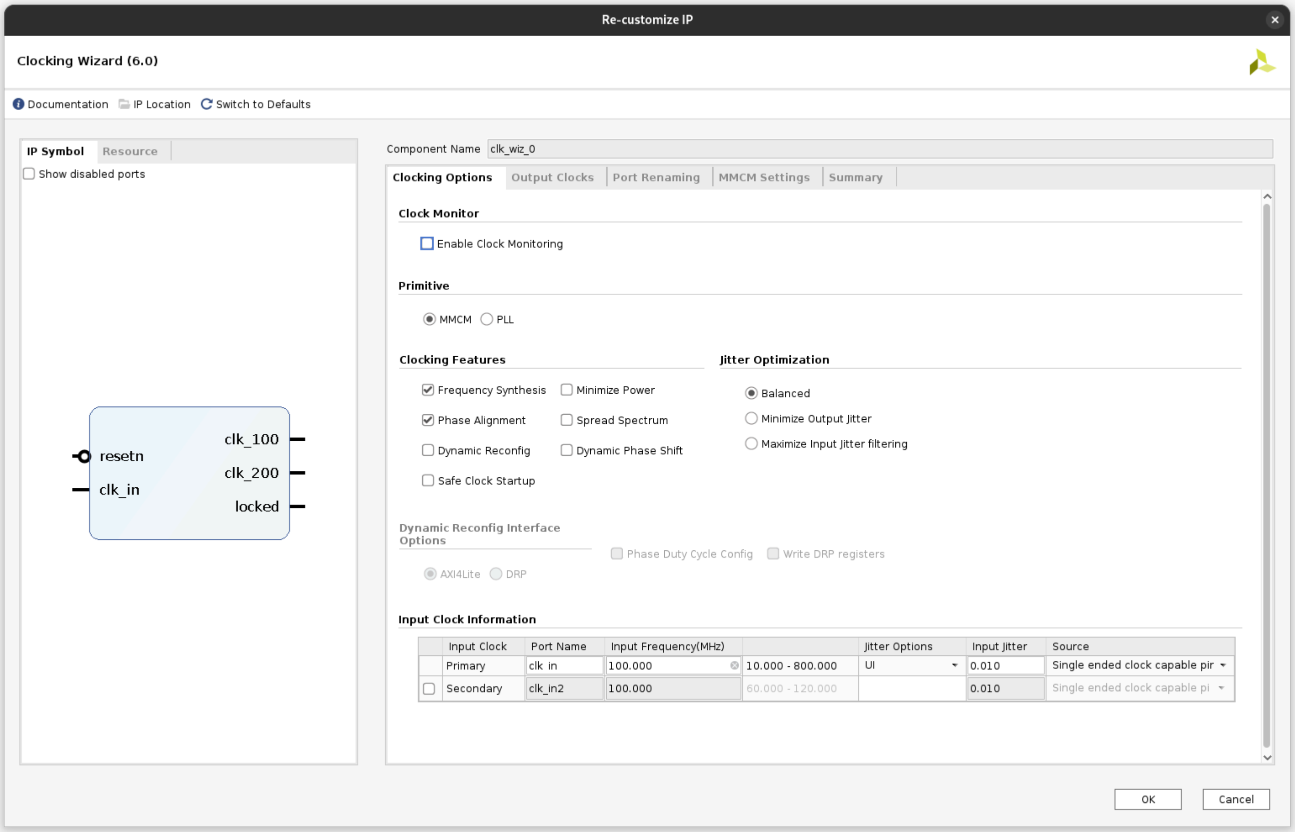
After a few seconds, a window should open that looks like this.


Manage IP - [/home/justin/alchitry/DDR Demo/cores] - Vivado 2022.1
File Edit Tools Reports Window Layout View Help
Quick Access
Default Layout
PROJECT MANAGER - xc7a35tftg256-1
Sources
IP (2)
clk_wiz_0 (29)
mig_7series_0 (146)
IP Sources
IP Catalog
Cores | Interfaces
Search:
Name AXI4 Status License VLNV
Vivado Repository
Alliance Partners
Alveo Card Management
Audio Connectivity & Processing
Automotive & Industrial
AXI Infrastructure
AXIS Infrastructure
BaseIP
Basic Elements
Communication & Networking
Details
Select an IP or Interface or Repository to see details
Tcl Console Messages Log Design Runs

| Name | Constraints | Status | Progress | Incremental | Run Strategy | Report Strategy |
|---|---|---|---|---|---|---|
| Out-of-Context Module Runs | | | | | | |
| ✓ mig_7series_0_synth_1 | mig_7series_0 | synth_design Complete! | 100% | Off | Vivado Synthesis Defaults (Vivado Synthesis 2022) | Vivado Synthesis Default Reports (Vivado Sy |
| ✓ clk_wiz_0_synth_1 | clk_wiz_0 | synth_design Complete! | 100% | Off | Vivado Synthesis Defaults (Vivado Synthesis 2022) | Vivado Synthesis Default Reports (Vivado Sy |

Note that you can see the two cores already in our project on the left.

You can right-click on the `clk_wiz_0` core and select *Re-customize IP...* to open up the wizard. Feel free to poke around the tabs to get an idea of the options available for future use.



Show disabled ports

Component Name: clk_wiz_0

**Clock Monitor**

☐ Enable Clock Monitoring

**Primitive**

[✓ MMCM] PLL

**Clocking Features**

☑ Frequency Synthesis ☐ Minimize Power
☑ Phase Alignment ☐ Spread Spectrum
☐ Dynamic Reconfig ☐ Dynamic Phase Shift
☐ Safe Clock Startup

**Jitter Optimization**

[✓ Balanced] Minimize Output Jitter Maximize Input Jitter filtering

**Dynamic Reconfig Interface Options**

☐ Phase Duty Cycle Config ☐ Write DRP registers

[✓ AXI4Lite] DRP

**Input Clock Information**

| | Input Clock | Port Name | Input Frequency(MHz) | | Jitter Options | Input Jitter | Source |
|---|---|---|---|---|---|---|---|
| | Primary | clk_in | 100.000 | 10.000 - 800.000 | UI | 0.010 | Single ended clock capable pir |
| ☐ | Secondary | clk_in2 | 100.000 | 60.000 - 120.000 | | 0.010 | Single ended clock capable pi |

Since this was already setup how we need, click *Cancel* for now and close the IP catalog.

Back in Alchitry Labs, you should see some text in the console about looking for cores. Everytime the *Vivado IP Catalog* is closed, Alchitry Labs does a sweep of the cores folder to see what changed.

Hooking up the `clk_wiz_0` core is simple.

```
// clock generator takes 100MHz in and creates 100MHz + 200MHz
clk_wiz_0 clk_wiz(.resetn(rst_n), .clk_in(clk))
```

The `clk_wiz` instance has two outputs, `clk_wiz.clk_100` and `clk_wiz.clk_200` which are 100MHz and 200MHz clocks respectively.

These clocks are fed directly into the `mig` controller. The `mig` controller then outputs a `ui_clk` which is intended to be used with its interface. This clock is 81.25MHz and is what will drive the rest of our design.

It is 81.25MHz because the DDR3 interface is set up to run at 325MHz with a 4:1 ratio.

Also notice that we are using the `mig.sync_rst` signal as our reset. Because of this, we don't need the `reset_conditioner` used in previous tutorials.

The reset button on the board will still work as the reset since it is used to reset `clk_wiz_0` which then resets the memory controller. This works since the `locked` output of the clock wizard goes low when it is reset. The `locked` output goes high when it isn't being reset and the clocks are stable. If you don't hold your circuit in reset when this input isn't high, you risk running into glitches as the clocks may be doing weird things.

With all of that we are now fully setup to use the core!

## The User Interface

The memory interface abstracts away a ton of the complexities of dealing with DDR3 memory, but the interface we get to it is still reasonably complex.

Check out this document from Xilinx that details all the information on the core. Skip to page 57 and look at the section labeled *User Interface*.

This section details what each of the signals we will be interacting with do.

The actual signals we need to deal with are a subset of the ones listed. See the two structs declared in the `mig_wrapper` component for the ones we need.

```
global Memory {
    struct in {
        addr[28],
        cmd[3],
        enable,
        wr_data[128],
        wr_enable,
        wr_mask[16]
    }

    struct out {
        rd_data[128],
        rd_valid,
        rdy,
        wr_rdy
    }
}
```

One thing to note is that while the `cmd` signal ( `app_cmd` in the Xilinx doc) is three bits wide, it only ever has a value of 0 (for write) and 1 (for read). Other values are used in different configurations of the core (for example, with ECC memory).

There are really three independent interfaces bundled together used to deal with the controller.

The first is the write FIFO. The controller will store data in a FIFO to be used in subsequent write commands. The signal `wr_rdy` is `1` when there is space in the buffer. You can write to the buffer by supplying data on `wr_data` and setting `we_enable` to `1` .

The signal `wr_mask` can be used to ignore bytes in `wr_data` . A `1` means to ignore the corresponding byte. To write the full 16 bytes (128 bits) set `wr_mask` to `0` .

Note that this mask doesn't change which bytes are sent to the DDR3 chip. It is used to control the *DM* lines which tell the DDR3 chip to ignore certain bytes. If you set `wr_mask` to `16hFFFF` and perform a write command, the full 128 bits will still be sent to the DDR3 but nothing will happen.

With data in the FIFO, you can issue a write command.

The command interface can be used when the signal `rdy` is `1` .

To issue a command, set `cmd` to `0` for a write command or `1` for a read command, supply the related address on `addr` and set `enable` to `1` .

If you issue a write command, the first value written to the write FIFO will be used as data.

If you issue a read command, once it has been performed, the value will be returned on the read interface.

The read interface consists of `rd_valid` , which is `1` when there is new data, and `rd_data` , which is the data. After sending a read command, you wait for `rd_valid` to be `1` which means the data you requested is on `rd_data` .

The reason for the three independent interfaces it efficiency. The command interface may be ready to accept more commands before the read data is fully ready. The write FIFO may also be willing to accept data while a read is being performed.

It is also possible to set up the core to allow it to reorder the execution of commands to improve efficiency. In this case you may be issuing many commands while waiting for the first read command to finish. However, to keep things simple, our interface configuration uses strict ordering (commands executed in the order given).

## Write and Read Example

In this next section, we will create a small state machine that first initializes the DDR3 to have sequential values stored and then reads them back displaying them on the LEDs.

Here's the full top level module.

```
module alchitry_top (
    input clk,              // 100MHz clock
    input rst_n,            // reset button (active low)
    output led[8],          // 8 user controllable LEDs
    input usb_rx,           // USB->Serial input
    output usb_tx,          // USB->Serial output
    /* DDR3 Connections */
    inout ddr3_dq[16],
    inout ddr3_dqs_n[2],
    inout ddr3_dqs_p[2],
    output ddr3_addr[14],
    output ddr3_ba[3],
    output ddr3_ras_n,
    output ddr3_cas_n,
    output ddr3_we_n,
    output ddr3_reset_n,
    output ddr3_ck_p,
```

First notice that we used the signal `mig.ui_clk` in the `.clk` block for the `dff` instances. This is the clock that the user interface of the memory interface is synchronized to. If you need to use a different clock for the rest of your design, yo have to implement some clock domain crossing scheme. It is easiest if you can mak your design work at the 81.25MHz of this clock.

If you look a little lower in the **`always`** block, you'll see the default values we ass to the `mem_in` struct on the `mig` module. The `wr_mask` signal is active low, meaning a 0 enables the byte. Since we will be writing all the bytes we can fix it to 0.

The other values we don't care about when the enable signals are `0` so they are set to `bx` .

The state machine is simple, it starts in `WRITE_DATA` where it writes a value to the write FIFO. Once the value is written, which is noted by `wr_enable` and `wr_rdy` both being `1` , it switches to the `WRITE_CMD` state.

In this state, it sends the write command. One thing to note is that the address associated with the write command is the *DDR native address*. This means that the first three bits should always be `0` since each DDR entry is 16 bits (the DDR3 on the Au has a 16 bit bus) and each read/write we preform is on 128 bit blocks.

This is easy to do by concatenating three 0's onto the end of our address.

```
mig.mem_in.addr = c{address.q, 3b000} // first three bits of addr are
```

The 128 bit operation size isn't configurable. Xillinx's memory interface uses bursts of 8 to increase efficiency. This is common practice when working with DDR memory. Also note that the DDR interface is clocked at 4x the system clock which means that you complete an entire burst in the span of a single system clock cycle (4x freq * 2 for double data rate = 8 values per system cycle).

The burst operation is actually supported by the DDR3 chip itself. The address you give the memory interface is sent directly to the DDR3 chip. If you don't set the last three bits to 0 weird things happen.

For writes, these bits are ignored and everything works as if they were 0.

For reads, the same 16 bit words in the burst will be read from same 128 bit block but in a different order. For example, if the last three bits are 0, then they are read in the expected order, 0, 1, 2, 3, 4, 5, 6, and 7. However, if you set the last three bits to 2 then it will read them in the order 2, 3, 0, 1, 6, 7, 4, and 5. See page 145 of this document for the full behavior. The memory controller is set up to use the sequential burst type.

It is generally best to just ensure these bits are always 0. The reason this feature exists is so that you can get a specific 16 bit value as soon as possible while still reading in a burst of 8.

Note that we are writing the address' value to each address. In other words, address 0 gets value 0, address 1 gets value 1, and so on. This means when we read these back in order, it'll look like a counter.

Once 256 addresses have been written, it switches to reading. The `READ_CMD` state issues the read command and then transitions to the `WAIT_READ` state.

In the `WAIT_READ` state, it waits for `rd_valid` to be `1`. It then uses the read value to set the LED's state.

Finally, it goes into the `DELAY` state to waste some time so we can see the LED value before it loops to `READ_CMD` again.

It will continue reading the first 256 addresses of the DDR3 over and over.

If you change what value is written in the `WRITE_DATA` state, it'll change what the LEDs show.

With this you should be able to build the project and load it onto your board to see the LEDs counting.

## Caching

You may have noticed that this example is incredibly wasteful. We are reading and writing full 128 bit blocks of data when we are only using the first 8 bits.

This could be fixed if we just combined 16 values into 128 bit blocks and wrote those to a single address. When we read them we could then store a whole line and iterate over each byte before reading in another line.

For our trivial example, it wouldn't be too hard to implement this directly. However, for more complex read/write patterns, this could be very difficult to do efficiently.

Luckily, there is a component in the *Component Library* that can help us with this.

Open the *Component Library* and under *Memory* select *LRU Cache*.

```
module lru_cache #(
    ENTRIES = 4 : ENTRIES > 0,
    WORD_SIZE = 16 : WORD_SIZE >= 8 && WORD_SIZE <= 128 && (WORD_SIZE ==
    AGE_BITS = 3 : AGE_BITS > 0
)(
    input clk,  // clock
    input rst,  // reset
    input wr_addr[24 + $clog2(128/WORD_SIZE)], // 24-28 bits
    input wr_data[WORD_SIZE],
    input wr_valid,
    output wr_ready,
    input rd_addr[24 + $clog2(128/WORD_SIZE)],
    input rd_cmd_valid,
    output rd_ready,
    output rd_data[WORD_SIZE],
    output rd_data_valid,
    input flush
```

This component is pretty complicated, but it can take the memory interface from t
*MIG Wrapper* component and give you efficient read and write interfaces of
selectable word sizes.

This cache is lazy and will only access the RAM when it needs to. That means you c
read and write to entries in the cache as much as you want without hitting the
memory. It will only access values in the external memory when it needs to free u

The cache presents independent read and write interfaces. This comes in handy when you have one section of your design doing only reads and another doing only writes. If you read and write the same address in the same cycle, you will read the old value.

You can configure the cache to have multiple entries (AKA cache lines). This can save a ton of IO for certain memory access patterns. For example, in our GPU demo project the rasterizer reads values from the Z buffer sequentially and writes values back sequentially at a slightly delayed time down the pipeline. This cache was used with two entries so that the reads and writes would each have their own cache lines to minimize fighting.

This cache attempts to approximate an LRU (**L**east **R**ecently **U**sed) cache policy. This means that when a new cache line is needed, the least recently used (AKA oldest) entry will be evicted.

The age of each entry is kept track of by adding 1 to its age counter every time a read/write is performed to any entry. The age counter can saturate if it isn't accessed in a long time.

Each time an entry is accessed, its age counter is reset.

The maximum age can be set with the `AGE_BITS` parameter. The default value is `3`, which gives a maximum age of 7. This way of keeping track of age isn't perfect and if `AGE_BITS` is too small, in some cases the cache may not act as a perfect LRU. However, this typically isn't an issue as it will always evict the oldest or a max age cache line.

Setting `AGE_BITS` to a large value will ensure that the oldest cache line is more likely to be removed but will incur a performance penalty.

## Cache Interface

The interface to the cache is pretty simple.

The addresses used are word address. This means you don't need to worry about zero padding anything.

You can set the size of the data word with the `WORD_SIZE` parameter. This can be set to 8, 16, 32, 64, or 128.

To write, you simply check `wr_ready`. If this signal is `1`, you can specify a value on `wr_data`, an address on `wr_addr`, and set `wr_valid` to `1`.

To read, you check if `rd_ready` is `1`. If it is, you set `rd_addr` to the address to read and `rd_cmd_valid` to `1`.

You then wait for `rd_data_valid` to be `1`. When it is, `rd_data` has your data.

When you perform reads, `rd_data_valid` is guaranteed to take at least one cycle from the request to go high. However, `rd_ready` may stay high if it is a cache hit.

That means you can stream multiple reads back to back with each result delayed a single cycle if they are all hits.

Cache misses will take longer for `rd_data_valid` to go high since it will need to actually fetch the value from the RAM.

If you have a cache in your design and are also reading values from another piece of your design, you may need to occasionally flush the cache to ensure that the values are actually written to the DDR3. For this, you can use the `flush` signal. When `flush` and `flush_ready` are high, the cache will write all dirty entries to the DDR3 memory.

## Cache Example

With the cache component in our design, we can use it to more efficiently use the DDR3.

```
module alchitry_top (
    input clk,              // 100MHz clock
    input rst_n,            // reset button (active low)
    output led[8],          // 8 user controllable LEDs
    input usb_rx,           // USB->Serial input
    output usb_tx,          // USB->Serial output
    /* DDR3 Connections */
    inout ddr3_dq[16],
    inout ddr3_dqs_n[2],
    inout ddr3_dqs_p[2],
    output ddr3_addr[14],
    output ddr3_ba[3],
    output ddr3_ras_n,
    output ddr3_cas_n,
    output ddr3_we_n,
    output ddr3_reset_n,
    output ddr3_ck_p,
    output ddr3_ck_n,
```

This new design performs exactly the same as before, but now we are using the DDR3 more efficiently. We are using 1/16th the memory as we were before without having to complicate our design. It actually simplifies the design since the writes don't require separate operations to write the data and command.

Note that `ENTRIES` is set to `1` since our access pattern is super simple and having more entries won't increase the number of cache hits we have (unless `ENTRIES` was set to 16 which would mean everything would fit in the cache).

We also set `AGE_BITS` to `1` since there isn't much of a choice which entry to evict when there is only one.

In this super basic use case, the cache component is overkill. However, the tools will optimize a lot of the logic out keeping it fairly efficient.

## Multiple devices

It is common to want to interface multiple parts of your design with the one memory interface. To make this easy there is a component in the *Components Library* called *DDR Arbiter* under the *Memory* section.

```
module ddr_arbiter #(
    DEVICES ~ 2 : DEVICES > 1
)(
    input clk,  // clock
    input rst,  // reset
    // Master
    output master_in<Memory.in>,
    input master_out<Memory.out>,
    // Devices
    input device_in[DEVICES]<Memory.in>,
    output device_out[DEVICES]<Memory.out>
) {
    enum State {WAIT_CMD, WAIT_WRITE, WAIT_RDY}

    .clk(clk) {
```

This module can be hooked up to the memory interface via the `master_in` and `master_out` signals. You then get `DEVICES` number of similar interfaces on the `device_in` and `device_out` arrays that can be hooked up to different parts of your design.

For example, in our GPU project, we have a section that writes frames and anothe that reads the buffer to display the frames on the LCD.

It is important to order your devices carefully since the device attached to index get full priority. If it never has idle bus time, it'll starve out all the other devices.

Register Interface