

Lab 12 – Processor Datapath

Introduction

In this lab you will be integrating your Arithmetic Logic Unit (ALU) from Lab 6 with the 2-read, 1-write register file from Lab 11 to create a datapath of a 4-bit microprocessor. In particular, the module you will be designing is a processor datapath, or *processor_dp*. The internal register size of this *processor* is 8 registers, each 4-bits wide – as defined by your register file from Lab 11. In addition to a skeletal processor datapath, it will have the display interface from the previous lab so you can easily read the contents of the registers.

When your design is implemented in your Basys3 hardware, the input control signals for this datapath will be from the switches and one of the pushbuttons. By toggling the switches you will *control* the processor datapath similar to the operation if a Finite State Machine controlled them. Thus, you will be selecting various arithmetic operations on the registers inside of your *processor*.

Even though we have 16 input switches and some pushbuttons, we need to share some of the control signals among the switches. The register file write addresses are shared by one of the read addresses, resulting in the data of one of the input operands being updated or written to, e.g., a processor operation: $R_x = R_x + R_y$. This is very similar to design tradeoffs made in the instruction machine code of microprocessors. In small microprocessors you do not see instructions of the type: $R_z = R_x + R_y$ – where all three address ports could be unique.

As in Lab 11 you will be controlling the reading, writing and displaying the memory contents by selecting various combinations of the switches and push buttons on the Basys3 board. With the additional ALU functionality, you will observe and update stored memory values (registers) of your choice, similar to how a controller within a microprocessor modifies its internal registers.

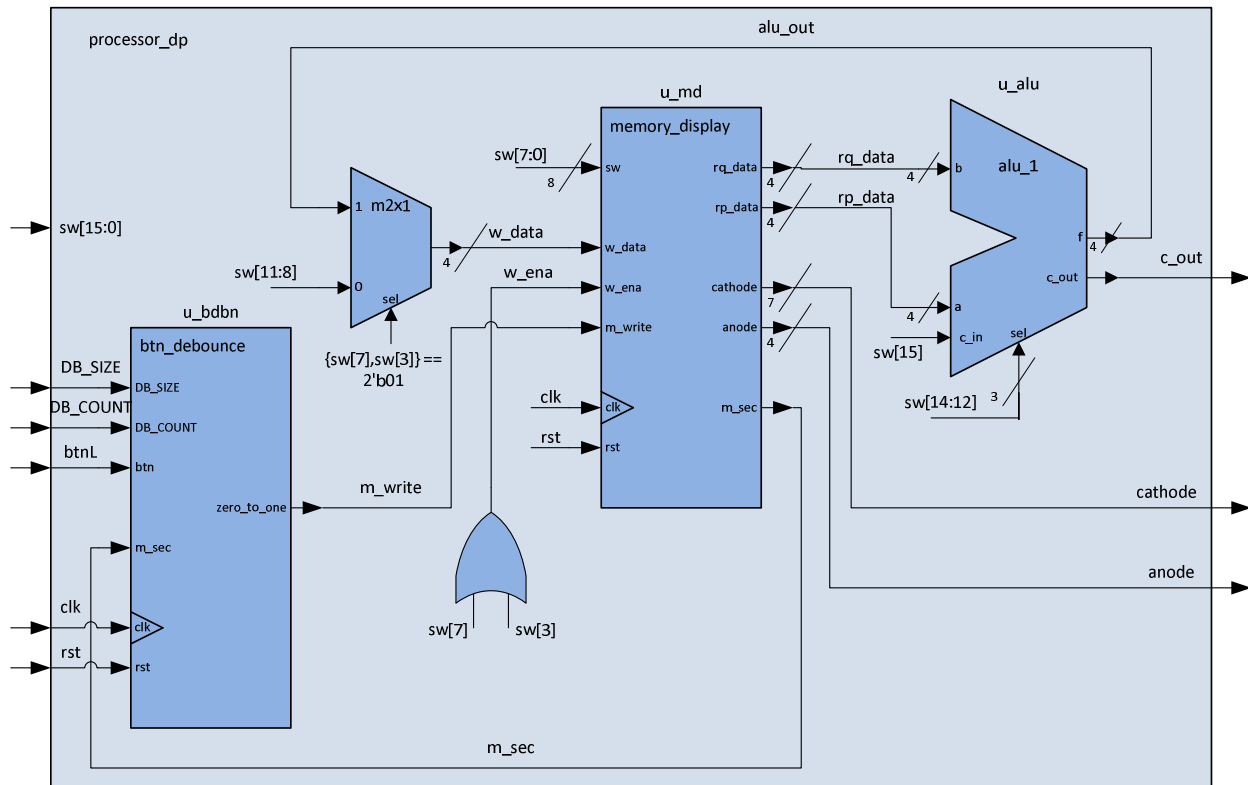
Detailed Specification:

A block diagram for the datapath module, *processor_dp*, is illustrated below. Note the sets of inputs and input parameters:

- a) the clock and reset (**clk**, **rst**);
- b) two pushbuttons
 - **btnC**->*rst*,
 - **btnL**->*controls write operation*,
- c) switches (**sw[15:0]**) for various data and control signals (a following table will document these).

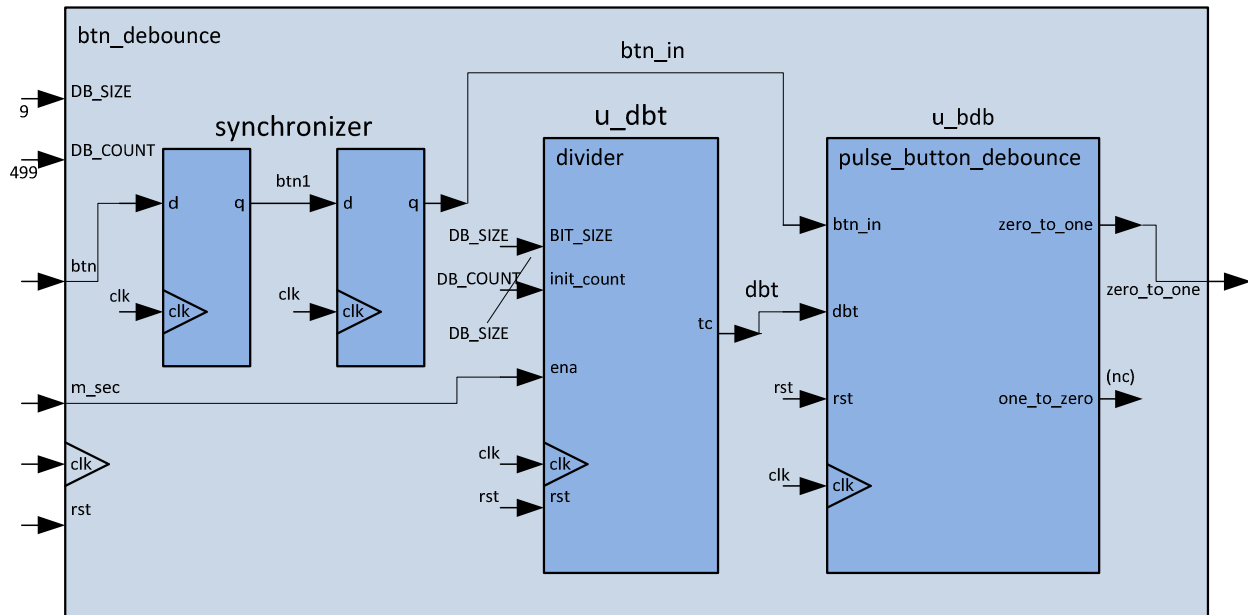
- d) two parameters (**DB_SIZE**, **DB_COUNT**) to determine the debounce time of the write enable pushbutton (**btnL**).

The output signals drive the 4 digit seven segment displays (**anode**, **cathode**), and a *carry out* (**c_out**) signal from the ALU (not shown it is connected to **led[15]**). A block diagram is shown below.



This top module instantiates three modules, **btn_debounce** (from the videos and provided in this lab), **alu_1** (from Lab 6) and **memory_display** (from Lab 11). The two output signals that were unconnected in Lab 11 (**rp_data** and **rq_data**) are now connected to the ALU as shown above.

Module **btn_debounce** creates a single memory write signal from a *bouncy* pushbutton signal, **btnL** (left pushbutton). Its block diagram is shown in the following figure.



This module will make use of your parameterized **divider** design, also from a previous lab. The millisecond (**m_sec**) signal from the **memory_display** module (see previous block diagram) is connected to its *enable* input (**ena**). The divider divides this by an additional 500 to achieve a 500 millisecond or *0.5 second* debounce time (**dbt** signal). Note this debounce time is adjustable by being defined in the two parameters: **DB_SIZE** and **DB_COUNT**. These parameters are brought up as input parameters to the processor datapath (**processor_dp**) module (previous figure).

The module, **pulse_button_debounce**, was also provided in a previous lab. It is basically a Finite State Machine (FSM) that provides a *one-clock pulse width* output for a *zero-to-one* and/or a *one-to-zero* push button event. The **zero-to-one** signal eventually drives the memory write signal (see previous **processor_dp** block diagram). Note also from the **processor_dp** block diagram that this is debouncing the left pushbutton or **btnL**, which functions as the write enable signal. In your hardware implementation *it must be pressed for ½ second to operate* (500 milliseconds).

The functionality of this design is summarized in the table below, with the differences from the design of Lab 11 highlighted. You will need this information when testing your hardware implementation.

Function: reset	
reset	center pushbutton (<i>rst</i> input signal in <i>memory_display</i>)
Function: write data to memory	
Inputs:	
Select write function	switch 3 off and switch 7 – on (<i>w_ena</i> input signal in <i>memory_display</i>)
Write address	3-bits: switches 2-0
Write data	4-bits: switches 11-8 (<i>w_data</i> input signal in <i>memory_display</i>)
Command: write data to address	left pushbutton (<i>m_write</i> input signal in <i>memory_display</i>)
Outputs:	
Digit 0	3-bits: hexadecimal, address to write to (switches 2-0)
Digit 1	4-bits: hexadecimal, data to write (switches 11-8)
Digit 2	(same as read function below)
Digit 3	(same as read function below)
Function: read data from memory	
Inputs:	
Select read function	switch 3 off and switch 7 – off (<i>w_ena</i> input signal in <i>memory_display</i>)
Read address, port p	3-bits: switches 2-0
Read address, port q	3-bits: switches 6-4
Outputs:	
Digit 0	3-bits: hexadecimal, address p
Digit 1	4-bits: hexadecimal, data read from address p
Digit 2	3-bits: hexadecimal, address q
Digit 3	4-bits: hexadecimal, data read from address q
Function: ALU operation – read data from memory while updating with ALU result	
Inputs:	
Select read register and update/write using ALU function	switch 3 – on and switch 7 – off
Read address, port p; and destination write address	3-bits: switches 2-0
Read address, port q	3-bits: switches 6-4
Command: update ALU result to register pointed to by port q	Left button
Command: ALU function	switches [14:12] (<i>alu_sel</i> – see your ALU design) <ul style="list-style-type: none"> - 000 -> f = a (no operation) - 001 -> f = a + b + c_in - 010 -> f = a + b with no carry - 011 -> f = a – b – c_in - 100 -> f = a – b with no borrow - 101-> f = a + 1 - 110 -> f = a – 1 - 111-> f = a & b (bit by bit logical AND)
Outputs:	
Digit 0	3-bits: hexadecimal, address p; and destination write address
Digit 1	4-bits: hexadecimal, data read from address p
Digit 2	3-bits: hexadecimal, address q
Digit 3	4-bits: hexadecimal, data read from address q

Design

You will be using 10 modules/files from your past labs: ***alu_1***, ***memory_display***, ***pulse_button_debounce***, ***rf_8x4_2r1w***, ***display_driver***, ***display_driver_dp***, ***display_driver_fsm***, ***divider***, ***svn_seg_decoder***, and ***anode_decoder***. These will be linked automatically for you in the simulation and synthesis scripts.

You will also use the button debounce module design provided to you from Lab 11: ***btn_debounce***.

A skeleton file is provided for the module shown in the block diagram: ***processor_dp***.

There is one design step:

- Design the ***processor_dp*** module based on the block diagram and design specifications earlier in this document.
 - instantiate the 3 modules (take care when instantiating the button debouncer – the parameters **MUST** be connected from the top level or your simulation will take hours to run – see example below)
 - creating an ***OR gate*** and a ***multiplexer***,
 - and connecting all the signals.

An example of instantiating the button debounce module while assigning its local parameters to the top level parameters:

```
btn_debounce # (.DB_SIZE(DB_SIZE), .DB_COUNT(DB_COUNT)) u_bdbn ...
```

Simulation/Verification

In the simulation view, there is one self checking test bench wrapper for the above module: ***tb_processor_dp.sv***. This file will read the test vectors from ***tb_processor_dp.txt***. Neither of these two files should be edited.

Verification step:

- ***processor_dp.sim*** is used to do a quick simulation the ***processor_dp*** module without saving the waveform data.
- ***processor_dp.flags*** is used to simulate with saving the waveform data. You should use this if you want or need to look at the waveform data. Be patient, because it's saving all of the signals it takes a few minutes to simulate.

The test bench has **136** test vectors that transition based on a change of the ***anode*** signals. Each set of 4 vectors are monitoring the ***anode*** and ***cathode*** signals so as to monitor the values

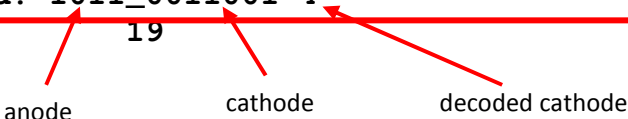
presented to each digit. The first **64** test vectors write to the 8-words of memory. In particular the *address-data* combinations are (in hexadecimal): 0-F, 1-D, 2-B, 3-9, 4-7, 5-5, 6-3, 7-1. Then various read/ALU write tests are performed as described in a following table.

The output from a successful simulation will be:

```
Match--loop index i:          0
Match--loop index i:          1
Match--loop index i:          2
...
Match--loop index i:          134
Match--loop index i:          135
Simulation complete - no mismatches!!!
```

If you have mismatches, the *expected* anode and cathode values are displayed, compared to their *simulated* values. To determine the input switch settings you will have to observe the timing diagram at the appropriate value of the index *i*. An example mismatch message could be:

```
Match--loop index i:          17
Mismatch--loop index i:        18; anode, cathode expected:
1011_1000000 0, received: 1011_0011001 4
Match--loop index i:          19
```



In this example the *decoded cathode* is the digit that would be displayed on the seven segment decoder based on the *cathode* pattern.

Below is a table that describes what the functions are for each set of test vectors. It is not an exhaustive testbench as all of your previous modules (such as the ALU) have been thoroughly tested when they were designed. (At this time the carry out signal is not being checked.)

Index i	Functional test
0-63	Write the <i>address-data</i> combination to the memory: 0-F, 1-D, 2-B, 3-9, 4-7, 5-5, 6-3, 7-1
64-79	Read the 8 words of memory and verify the pattern, two addresses at a time
80-87	Add: $R0 = R4 + R0$, result should be 6
88-95	Add with carry: $R0 = R4 + R0 + c_in$, result should be 14 or E
96-103	Subtract: $R2 = R2 - R6$, result should be 8
104-111	Subtract with borrow: $R2 = R2 - R6 - c_in$, result should be 4
112-119	Increment: $R3 = R3 + 1$, result should be 3
120-127	Decrement: $R5 = R5 - 1$, result should be 10 or A
128-135	Logical bit AND: $R6 = R6 \& R1$, result should be 1

Correct your mismatches before you implement the design in hardware.

Synthesis

Generate a bit file as you did in past labs (right click and Run on: *lab12_top_io_wrapper.tcl*). For this lab it will be called: ***lab12_top_io_wrapper.bit***. Copy it to your memory stick to load and run on your Basys3 board.

Using the functional table shown previously, select the switches and push buttons to enable some of the operations. In particular, test your design by writing to the 8-words some random 4-bit numbers of your choosing, i.e., part of your TUID, part of your phone number, etc. Complete the table below with the data. Then select the read mode and see that you can read back all of the stored data – as in Lab 11.

To test the ALU functionality, set ***switch 7 – off*** and ***switch 3 – on***, as described in the instruction table. Select two read memory locations (registers) with the other 6 switches. Select an ALU operation you want to perform (***switches 14-12***), and press the ***left pushbutton*** down **for ½ second**. You should see the destination register (***rp_data*** on ***digit 1***) change its value, as a result of the ALU operation. Try it for several different registers and all seven of the ALU operations (different combinations of switches 12-14).

Because of the debounce circuit, you cannot press the left pushbutton too quickly (at least ½ second down and ½ second up).

Use the table below to record your results. There are extra columns for the arithmetic operations.

Memory Address	Data		
arithmetic operation			
0			
1			
2			
3			
4			
5			

6			
7			

Extras:

We have used many different modules from past labs. Here is a screen shot of the hierarchy of the completed design from a past tool and from the current waveform viewer. Can you identify what modules are used from what labs? Note also the depth of the hierarchy. The seven segment decoder and anode decoder are pretty deep in the hierarchy.



You can change the speed of the pushbutton presses by changing the parameters in the debounce timer. These are declared at the top level hardware wrapper and the top level test bench.

The divide time of the debounce timer is different for the hardware implementation versus simulation. To simulate a half second delay for each of the different *write enable* button presses would take a significant amount of simulation time (maybe hours). In order to speed this up, the divide value is pulled up (as a parameter) to the top level so that it can be changed and reduced to a smaller value for simulation.