E533 Laboratory #5

Pipelined Datapath Design for the ARM-compatible Processor

Instructor: Young Cho - youngcho@isi.edu

Review: Thorough Testing of the Hardware

Most of you should've gone through these steps to get your system working. Nevertheless, I would like to re-emphasize the steps in this lab to get your system working. One of the first things to do is insert a passthrough ID.v into the NetFPGA reference NIC or Router package, which simply connects all of the modules' inputs to outputs using a "wire" type variable. Compile the source as you did in Lab 4 and generate the bitfile. Download the generated bitfile to the FPGA node using Linux command-line tools such as scp. Reconfigure the FPGA with the generated bitfile and make sure that it is functionally the same as the reference bitfiles. I recommend that you also develop, simulate, integrate, then test a logic analyzer built using a block RAM. You should be able to use your register interface to control the function of the logic analyzer to record any internal signals and dump it out to the server using the script. After this, incrementally add components from your mini-IDS design a little at a time to compile and test the proper functioning of your design.

Load your custom-generated bit-file using nf_download. Make sure the interfaces are properly configured, then start the rkd. Make sure the nodes can ping each other. In the lab4_mini_ids_src.zip file, you'll find a Perl script called `idsreg`. Copy this file to the FPGA node and make sure it is executable (you can use scp to copy it from your computer to the nodes).

Examine the script to understand what it does. The script simply constructs register bits out of what a developer might assign as commands and data. Then the register's contents are transferred to NetFPGA for the hardware design to interpret according to the developer's rules.

Once you understand what the script is doing, modify it to suit your needs. One of the first things you will need to do is change the address in the script to point to the NetFPGA compiler-assigned register address(es) found in the generated files in the lib folder. Modify the script to write and read to/from your registers

One simple design might be to write an 8-byte value into a mapped "write" register, wire up the hardware to reverse the byte order, and write the new 8-byte value into another mapped "read" register that you can read using the script.

Using your idsreg, you might want to try the following. To ensure these commands work correctly, you may need to modify the script to support your underlying hardware design.

```
> idsreg reset
> idsreg pattern ABCDEFG
```

ABCDEFG might be a 7-character string of your choice.

```
> idsreg matches
```

At this point, you should see count = 0x00000000

Start an iperf server on each node. Start three iperf clients on each node. The clients should send their packets to the other nodes. For two of the iperf clients on each node send "good" packets (i.e. the packets do not contain the string). On one iperf client use the -I option to include the string you chose above. This will create "bad" packets.

You should observe that the packets from your "bad" iperf clients do not get through. Observe the difference between TCP and UDP modes. In TCP mode, your "bad" client should connect, but no data packets should arrive. In UDP mode, no packets should reach the server from the "bad" client. You can use tcpdump to verify this fact. Include a snippet to show that each server is only receiving packets from two clients. On the FPGA node, run idsreg matches, and you should see the number of dropped packets. Continue with the above process until everything works as intended.

Answer the following questions in your report:

1. Explain in brief how the mini-IDS works and how it interacts with the other modules in the NetFPGA. Be **clear and concise!** Write as if you are describing the mini-IDS project on a webpage where other NetFPGA developers will read about your design.
2. Explain the pattern matching algorithm.
3. Draw a high-level design of the user_data_path.v and ids.v Verilog files. The figure should depict the communication between the components.
4. A major part of the assignment will be in demonstration of your system. Please complete the process of going through your design in your YouTube demo video.

Part 1 – Extending Synchronous Adder into ALU

You must extend your synchronous 8-bit full adder into a synchronous Arithmetic Logic Unit (ALU) that is at least 32-bit wide. You must include add, subtract, bitwise AND, bitwise OR, bitwise XNOR, compare, and logical shift functions. Other useful network-related functions may be substring comparison and shift-then-compare. You may use any combination of schematics, Verilog, Core IP, and previous projects. Just do not copy someone else's work. If we find out you copied someone else's work, you will receive zero for the entire laboratory.

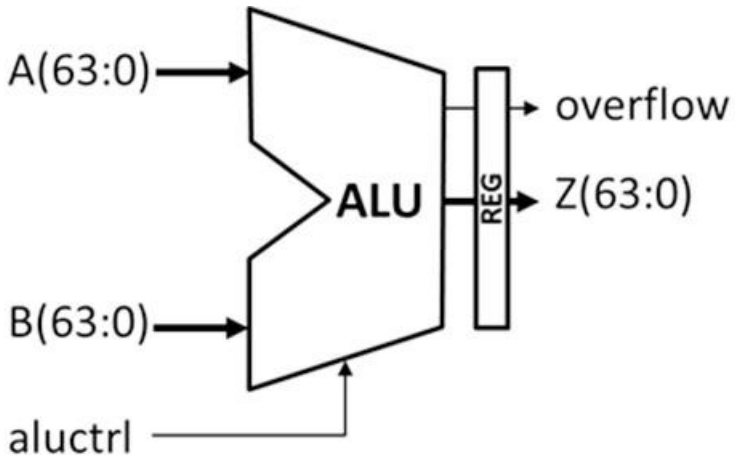

Figure 1: 64-bit Arithmetic Logic Unit

Make a table for the ALU control and what each mode represents (this table should be unique for each team). Simulate your circuit and verify the design's correctness. Take screenshots of your simulation results, include a description and the table, and include them as part of your write-up.

Part 2 – Building Register File and Memories

You are to build a register file for your processor. It must have at least four 32-bit registers (64-bit is recommended, since the NetFPGA datapath is 64-bit wide). A typical register file can be constructed with two read ports and one write-back port, as shown in the following figure.

Simulate your design and verify its correct functionality. Include screenshots of your simulation results and explanations in the report.

Use Core IP to generate a BRAM-based dual-port synchronous memory with 64-bit-wide data I/O and 256 entries. Check that you are using a single BRAM. This will be the data memory for your processor for now. Also, generate a BRAM-based single-port synchronous memory of 32-bit width and 512 depth, or 16-bit width and 1024 depth (depending on your ISA). Check that you are using a single BRAM. This will be the instruction memory for your processor. Test your memory modules for correct functionality. You do not need to include any screenshots for these units.

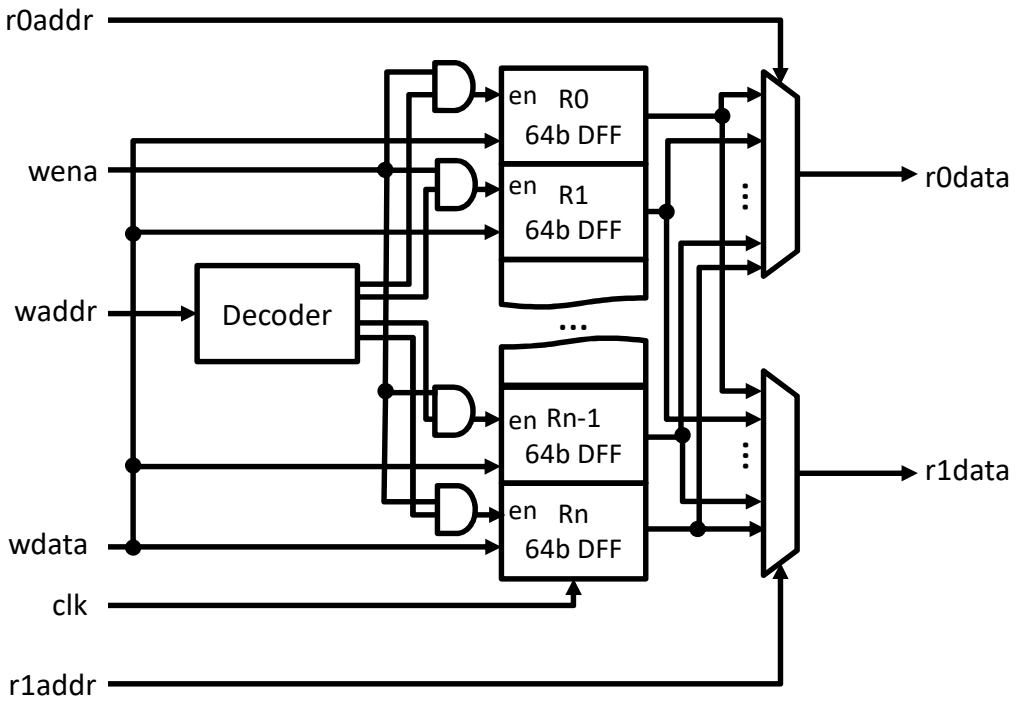

Figure 2: 64-bit Register file with two read ports and 1 write-back port

Part 3 – Building Pipeline Datapath

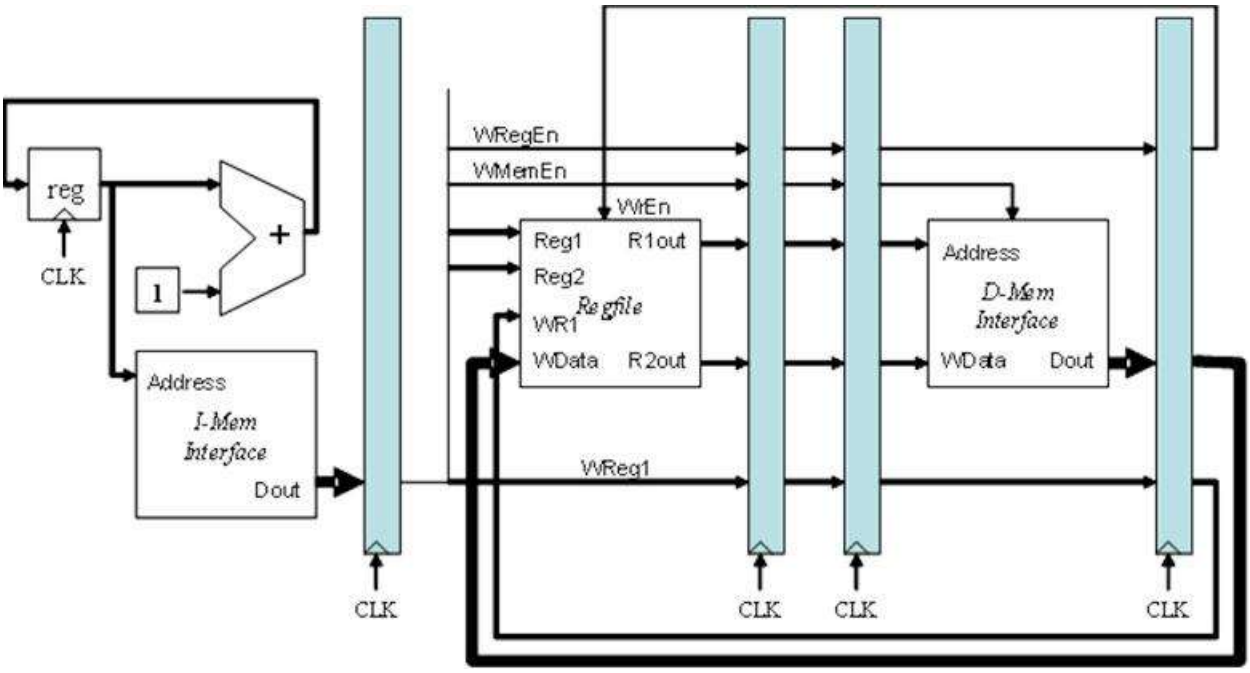

Figure 3: Skeleton Pipelined Datapath

The datapath is a framework on which your pipelined processor will be built. The following figure shows a datapath you can build to test your memory interface and register files without a controller. Eventually, you will add the ALU from part 1 and Multiplexers to make a full pipelined processor, but we will not do that for now. For this datapath to function correctly, you need simple and dumb Instructions. These instructions can be loaded onto the memory before the processor executes.

Instruction Format

WMemEn (1bit)	WRegEn (1bit)	Reg1 (3bits)	Reg2 (3bits)	WReg1 (3bits)	Unused (rest)

Create a small program in bit vectors that first loads the data from D-Mem into one register and then loads it into another register. Then store the value of Reg2 at the address in Reg1. The following may be an example of your D-Mem content and a test program:

Data Memory

Addr	Value
0	4
1	X
2	X
3	X
4	100

Instruction Memory

Addr	WME	WRE	REG1	REG2	WREG1	Comments

0	0	1	000	XXX	002	//Load Address 0 (Reg0=0) of D-Mem to Reg2
1	0	1	000	XXX	003	//Load Address 0 (Reg0=0) of D-Mem to Reg3
2	0	0	XXX	XXX	XXX	//Nop
3	0	0	XXX	XXX	XXX	//Nop – value 4 written into Reg2 by now
4	0	0	XXX	XXX	XXX	//Nop – value 4 written into Reg3 by now
5	1	0	002	003	XXX	//Store 4 into Mem address 4

Preload the memory modules with the values above and simulate the pipeline execution using the tools. Fill the remaining I-memory with 0s to prevent the datapath from corrupting your results. Take screenshots of your simulation results, include a description, and include them as part of your write-up.

Part 4 – Integrating the Pipeline into NetFPGA

You will place your design in the user_datapath, and all of your interactions with your datapath should be through the software/hardware register interface. As with the Mini-IDS lab, you should start with the reference router design and connect the input and output pins of the user_datapath.v isolating the processor from the router design.

You need to design an interface to program your data memory and instruction memory using software/hardware registers. You are free to do whatever you want to make this happen. One way to achieve this is by using an address map. Your I-memory and D-memory can be assigned address spaces (i.e., I-memory could span addresses 0-511, and D-memory could span addresses 512-768). Your interface may have registers for command, address, and data. Then, whenever you assign values to the address and data registers along with a write command, your interface module can write the data to the corresponding memory. You should also build an interface to read values from memory. This interface can be used to verify the correctness of your design.

Include a description of your interface module and the block diagrams in your report. In addition to your report, you are to demonstrate the working of this datapath in your YouTube demo video

Submission and Demonstration

- Draw a high-level design of the datapath. The figure should depict the communication between the components.
- Include the following in your report:
  - Screen Capture of Schematics
  - Generated Verilog Files
  - GitHub records and descriptions per team member
- A major part of the assignment will be in demonstration of your system. Please complete the process of going through your design in your YouTube demo video.