Design of the ALU

ECE/CS 3710 - Computer Design Lab

Lab 1 - Completion and demo to the TA by Tuesday Sept 8. Lab 2 will be assigned and discussed on Thursday Sept 3 (yes before Lab 1 is over).

I. Overview

The instruction set of the CR16 uses standard 16-bit integer ALU operations. If we assume that the decoding has been done (actually you will design the decoder in a following lab), what you need to build to execute each instruction is an ALU to compute the result, and a register file to provide arguments and hold results. This lab requires you to design the ALU. In the next lab, you will design the register file and integrate it with the ALU.

As an aside, it is interesting to note how much of the machine can be designed before knowing the complete picture of what the machine will look like. There are a set of basics that are common to a number of different designs. Building these basic components (memory, register file, ALU, etc.) do not really require knowledge much more than the word size and the overall plan. Filling in the details later will determine the precise character of the final machine. Most VLSI design systems, for example, have large libraries of these generic parts which can be used to design a wide variety of different machine types.

II. Execution Datapath

In class, we have looked at the baseline instruction set of our processor. Looking at the Baseline instruction set you should notice the following:

1) The word size is 16 bits. Therefore, although it is not absolutely essential, it would be natural to have a 16 bit ALU. It is not required because you could clearly perform the ALU operations sequentially, using a smaller ALU. The 16-bit result could, for example, be generated using an 8-bit ALU and two time-steps. Going to the extreme, you could use a single-bit ALU and iterate 16-times for each ALU operation. Anyway, getting back to the point, your ALU should probably be 16-bits wide to match the data word size.
2) The instruction set supports both *ADD* and *SUB* instructions, so your ALU should also support these operations. Moreover, operations can be performed over *unsigned* as well as *signed* integers. Reading the CR16 handout carefully you will note that *signed* numbers are represented as *two's complement* numbers. So, your ALU should operate on *two's complement*, and can use this representation for implementing subtraction. The main difference between a *two's complement* ALU and *unsigned* ALU is in the way subtraction is encoded, and also in the way condition codes are generated. The condition codes required by the Baseline instruction set are defined here and also in the programmers reference manual:
   - **C** – The *Carry bit* determines whether a carry (or borrow) occurred after addition or subtraction. 0 means no carry or borrow, 1 means a carry or borrow occurred.
   - **L** – The *Low flag* is set by comparison operations. L is set to 1 if the Rdest operand is less than the Rsrc operand when they are both interpreted as *unsigned* numbers. You can use this for two's complement too.
   - **F** – The *Flag bit* is a general flag used by various operations to signal exceptional situations. In particular, arithmetic operations use the F bit to signal arithmetic overflow (this is often called the V bit on other machines). Let us use this as the overflow flag. (You know when to set/reset this bit).

- **Z** – The *Z bit* is set by the comparison operation. It is set to 1 if the two operands are equal, and is cleared otherwise. Here is one extension: The moment the ALU result is 0 (say, due to AND with 0s), why not just set it? Why restrict it to just comparison operations?
- **N** – The *Negative bit* is set by the comparison operation. It is set to 1 if the Rdest operand is less than the Rsrc operand when both are considered to be *signed* integers.

3) So to conclude, we should do the following arithmetic operations:
   - Lets treat data as 16-bit words (baseline). Use of 8-bit ALU operations can be implemented as an extension, if you wish.
   - Let us implement both unsigned and signed operations (ADD, compare, shifts). When it comes to subtraction, lets use 16-bit two's complement.
   - Design issue: Unsigned compares are easy to implement. You need to think about how to implement *signed*, i.e. *two's complement compare* operations. Design of two's complement compares in hardware can be beneficial, as it can make software design much easier.
4) The processor supports logical operations of *AND, OR* and *XOR*. Like the arithmetic operations, these can either take two *registers* as arguments, or one *register* and an *immediate* value. **Note:** Baseline does not have a NOT operation; maybe you would want to include it?
5) The Baseline instruction set has one shift instruction: *logical shift*. *As a baseline, we will implement 1-bit Left and 1-bit Right shift*.

   An obvious extension of the baseline set would include an *arithmetic shift*; and I would encourage you to implement arithmetic shift. These instructions have an *immediate* field as part of the instruction. In the full processor design this is used to indicate how many places to shift. A value of 1 in the *immediate* field shifts the argument register by 1 place to the left. A value of -3 indicates a shift of 3 places to the right. In our Baseline processor, you can assume that all shifts will be by one only. The only legal values for the *immediate* are 1 and -1 to indicate left and right shifts. You are, of course, free to augment this in your processor.
6) *Branch* and *jump* targets are computed in one of two ways: either as a displacement from the current PC, or to an absolute value held in a register. In the case of the displacement, the current PC value must be added to the immediate value in the instruction and written back to the PC. In the absolute case, the value in a register should be written directly to the PC.

   There are two ways to implement *Branch, Jump* operations. One of them corresponds to using the ALU to compute the the target address. The other way would be to design a dedicated ADD/SUB unit for branch target address computations. I would urge you to think about both options and figure out which one would you prefer to design. In class, we were thinking in terms of the second option. So, I would say, *let us omit Branches for now*, and we'll come back to them later - perhaps in the third lab.
7) *Load* and *store* instructions need to use values in registers, possibly added to immediate values in registers, to compute memory addresses. In class, we were discussing that load/stores can pass through the ALU; ditto about *move* instructions. It is your choice how you want to handle these.

Looking at these constraints it is possible to do a general layout for the execution datapath of the machine. The important thing at this point is not to get everything exactly right the first time, but to make sure that you have not made anything impossible that you will need to do later. The set of control points in this datapath (function codes, mux select signals, latch enables, etc.) will be controlled by the control state machine and the decoded instruction later.

A. ALU-bus architecture

The ALU should be able to read two register's contents and perform an operation on them. However, the ALU must also be able to operate on a register and an immediate value (ADD-immediate for example). These immediates will need to be *sign-extended* before going into the ALU, and there are a couple different lengths of immediates to consider. Computing with immediate values in Verilog is easy; just use a 16'b-value as one of the operands. However, how does the result get synthesized? You may want to see the synthesis result. Specifically, w.r.t. *immediate* instruction, your design choice is to either: i) model immediate instructions implicitly within the always()-block in Verilog, and let the synthesis tool generate the immediate data-input selectors; or ii) explicitly provide immediate data-input via MUXes/BUFs to the ALU.

Some designs (or designers!) model shifters outside the ALU. In today's world of high-level abstractions and synthesis, such excessive partitioning may be unnecessary. But, if you insist on modeling the shifters outside your ALU, then you need to address the following issues: "It is possible that a register value might be sent to the shifter instead of the ALU. So, the output of the execution unit might come from either the ALU or the shifter. Sounds like we may need MUXes (tri-bufs?) on inputs/outputs."

The ALU for the first lab is a stand alone unit. In the next lab, you will integrate the register files with the ALU. As a **guideline**, here is how I am thinking of organizing my CPU. Note that the diagram below is just a guideline, not a mandatory architecture. *Do not take this block as the required organization!* This is just an example of how things might be organized. This organization might help you in terms of designing a test-bench around your ALU, and perhaps it might even expedite the register file integration.

If you have a different organization in mind, by all means use that. Just make sure that it is able to compute all the results that will be required by the given instruction set! By being clever you may be able to reduce the number of muxes, or combine units together, or modify things in any number of interesting ways.

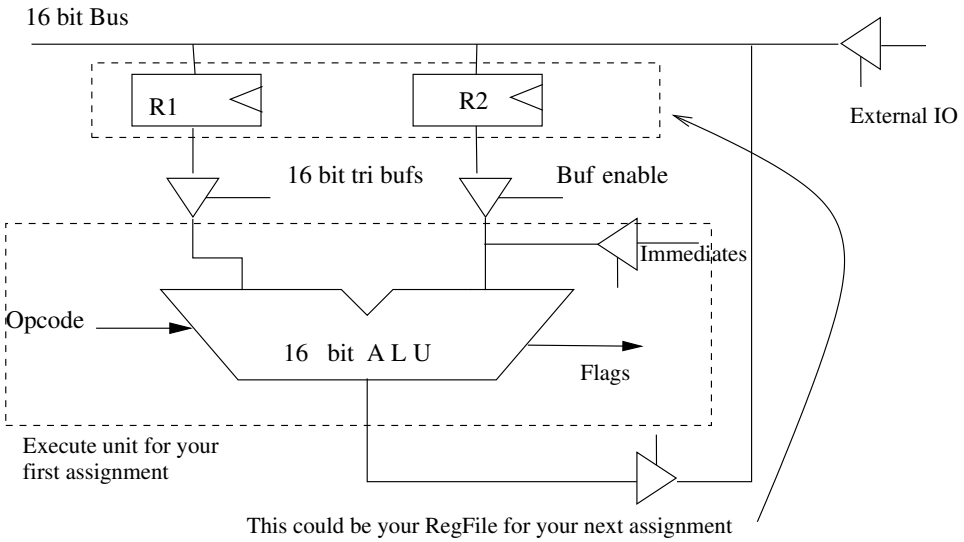

Fig. 1. Sample Block Diagram of the 16-bit ALU Datapath Organization. The architecture shows Buffers, but may instead use Muxes.

One approach to designing this ALU datapath is to look at every single instruction that you are interested in (i.e. the Baseline instruction set, plus the exception if you are interested in that), and check that each one can be accommodated by your datapath.

III. The Assignment

To conclude, we will implement the following baseline instructions from the ISA sheet, which is also uploaded on the web. Boolean operations are straight-forward. For arithmetic, we will use unsigned as well as 2's complement representations. Left/Right Shifts (1-bit shifts only) and moves should also be a part of the ALU. Comparison operations, both signed and unsigned, should also be implemented. Anything else you want to implement through the ALU, feel free to discuss it with me or the TA.

We will do: ADD, ADDI, ADDU, ADDUI, ADDC, ADDCU, ADDCUI, ADDCI, SUB, SUBI, CMP, CMPI, CMPU/I, AND, OR, XOR, NOT(?), LSH, LSHI, RSH, RSHI, ALSH, ARSH, NOP/WAIT. Anything else you have in mind?

Remember that your ALU must generate condition codes that will be latched into the processor status register (PSR). Your ALU should generate all the Baseline condition codes. Also pay attention to which instructions update the condition code register! Not all ALU instructions update the register, and even those that do, do not all update it in the same way! Some instructions only set some of the bits, and some instructions do not set any of the bits. Not even all the ADD instructions set the PSR. So, you need to have control over whether the PSR is updated or not. The decoder can decide which PSR bits are updated. You may use a dedicated 8-bit register as the PSR, and may or may not implement it as a part of the register file. As far as the (combinational logic) ALU is considered, its job is to compute the "flag" signals – how and where are these latched, we can do that later.

Design: You will design the whole unit in Verilog. Think structurally. Try to partition your design into simple computational units.

Simulation, Synthesis & Testing: Simulate your design exhaustively. Once it is functioning correctly, synthesize it, and generate the netlist. You should re-simulate the synthesized netlist to ensure bug-free synthesis and to validate timing. Then map the design and test it on the board. You can use the on-board switches and the 7-segment/LCD displays to validate the result. Now, here is a problem: On this FPGA board switches are few and data size is 16-bit. How will you test your implementation. Some data-hard-coding might be required. I will provide you some guidelines on how to demo the hardware by the end of this week.

At least for simulations, the test bench should be as exhaustive as possible to get high coverage and should have self checking capability. The same test bench allows you to verify the circuit and then validate the circuit after you make changes. A good test bench file will also be well commented.

Optimization: Ideally, we want the circuit to be both fast and small. In multi-level logic, it is not easy to do both. I would suggest you synthesize/optimize your circuits once each for area and speed, compare the results, and make your own judgement! In particular, how many Xilinx CLBs are used? Which paths are longest paths? You already know how to generate the synthesis reports.

Deliverable: Make a list of all the Opcodes you are implementing, what is the instruction encoding, what addressing modes have you covered so far, a etc. Generate the above information, collate it into a table - a list of what you implemented, keeping a list of what possible extensions might be needed later-on, which ones would be straightforward and all that - this way you would have documented the development of your datapath.

I'm NOT going to ask you for a written report for the ALU design. However, you will have to prepare a report after lab 2, when register files are integrated with your ALU. Because there will be a lot of overlap/interfacing between the two labs, I would rather you prepare a comprehensive report (with objectives, what did you implement, how did you go about doing it, synthesis and simulation results, area/delay reports, etc.) after lab 2. For this lab, you should demo both hardware and verilog to the TA. ALU is combinational, so it should be easy... (or maybe I should say easier) that other modules.

Final Demo: Demonstrate a functioning circuit (on the FPGA) to the TA latest by Sept 8.

Preserve your Verilog Code, you'll be doing a lot more with it in the next assignments. If you have any questions, feel free to discuss your design with me or with Steven. Hope you'll have fun with this lab.