Department of Electrical and Computer Engineering
State University of New York at Stony brook

ESE 555 Advanced VLSI Systems Design (Fall 2009)

CAD Assignment 4: The ALU

Assignment

To design a 16-bit ALU which will be used in the datapath of the microprocessor. This ALU must support two's complement arithmetic and the instructions in the baseline architecture.

Description

The Arithmetic and Logic Unit (ALU) is the heart of your processor. The minimum set of instructions your ALU needs to support consists of (i) ADD, (ii) SUB, (iii) CMP, (iv) AND, (v) OR, and (vi) XOR.

Your ALU must support both register-based operands and immediate operands. You can implement all the above instructions with the adder as the basic building block. A two's complement subtracter and a comparator can be derived from the adder structure. Two's complement subtraction (A-B) is implemented by adding A, B_bar, and 1 (Carry_in = 1). Compare functions (GE, LE), which are used to set processor flags for conditional branches, can be implemented with a subtracter and the information from the most significant bit (sign bit). Implementing "Less Than" is a little more complicated because of overflow problems. To detect a zero output, you need a NOR tree to verify that no result bit is a 1.

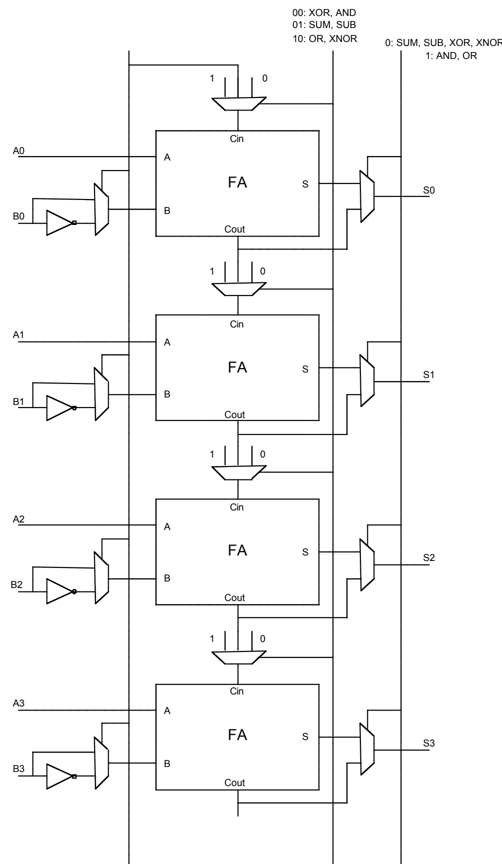

Figure 1: Merged Arithmetic/Logic approach.

Your ALU must perform six different functions; this could be accomplished as shown below, with the adder perform the logic functions as well as arithmetic functions. This shows that appropriate switching of the carry line between adder elements will give the ALU logical functions. An example block diagram is shown below. Any of a variety of 16-bit adder designs could be used in your processors. Whichever design you choose to implement, you would first build a 1-bit cell. Note that any number of such elements may be cascaded to form an adder of desired width.

Procedure

Your ALU must be pitch/bitslice width matched to your register file. In this assignment, you are not required to implement the logic to set processor flags (zero, overflow, carry, etc.), but you should have in mind the requirements so that you do not have to rework your design later (e. g., how will you detect an overflow?).

1. Schematic Design
General-purpose processors usually use designs that are not shown here because they are more complex. The text and other reference books have many examples, which you will want to know about, but which you probably do not want to implement in this assignment.

As you design your ALU, think about the overall microprocessor organization. Distribute buses and control lines the same way you did in the register file. Leave space for any signals that will be needed to interface to logic for checking overflow, setting processor status bits (flags), etc. You will also need a sign-extend module to convert immediate data to 16-bit data for the ALU.

2. Layout
The ALU is the key block of your datapath. Careful optimization of the ALU is of utmost importance. Since the critical path is on the carry stage, it may be advantageous to size the transistors in that stage accordingly. However, transistors in the sum stage can be minimum size. Arrange the transistors switched by the carry-in signal to be close to the output. This will reduce the body effect on carry-stage transistors.

Since your ALU performs only six functions, three control lines would be enough; however, you will find it easier to design a cell which has more control lines. You can leave some of the control points open at this stage of the design. If you do leave them unconnected, label them otherwise you will get errors in LVS. Be sure to run the buses over your cells. Pitch/bitslice width matching with other datapath modules is very important. Though you don't need to implement the datapath right now, think about the various sources of inputs to your ALU. They could be from the register file or the program counter or from the instruction register (displacement). A little thought about floorplanning of your datapath organization in advance will help a lot in getting a good final layout.

3. Design Verification
Do DRC and LVS to verify the layout. Generate report files.

4. Analog Simulation
Extract the parasitic capacitances and then use Spectre to get the worst case delays for the 1-bit cell (or the 16 bit ALU if you can). Since it is time-consuming to simulate the 16 bit ALU in Spectre, think how best you can simulate a 1-bit ALU and extrapolate the delay information for the carry_out of the 16 bit ALU.

Comments

° You can implement the ALU any way you wish. Remember that it is a datapath element. You can choose any select-line set-up that you wish (i.e. you are not restricted to any type of encoding).
° You are allowed to add instructions to the basic instruction set, but do not remove any.
° Keep in mind that you are going to need to implement some Processor Status Register (PSR) bits. You implement the PSR later in the semester. However, some of the groups might wish to place the PSR on the datapath. If this is the case, you might want to implement the PSR in this CAD assignment since you are

familiar with the ALU and time will be tight later in the semester. See the PSR Description provided at the end of this assignment for more information.

Requirements

All files should be placed in your group directory in the directory cad4.
° 16-bit ALU (alu16) schematic and layout plots. You should also have the schematic and layout for the 1-bit ALU (alu1).
° Spectre traces for the 16-bit ALU. These should demonstrate all ALU functions.
° Spectre output showing the longest rise and fall delays for each output of the 1-bit ALU. Explain the delays; i.e., which path was critical in determining the delays.

Deadline

You need to turn in CAD 4 by Thursday, October 29, 2009.

Appendix:

PSR Description
The Processor Status Register is a 16 bit scannable register that holds information pertaining to the current state of the microprocessor. The assignments of bits in the PSR is shown below.

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
R	**R**	**R**	**R**	**I**	**P**	**E**	**0**	**N**	**Z**	**F**	**0**	**0**	**L**	**T**	**C**

Bits 12 to 15 are "Reserved" which means that these bits should not be written or read. Bits 3, 4 and 8 may be hardwired to zero. The bits **N**egative, **L**ow, **Z**ero, **C**arry and **O**verFlow are set by the various arithmetic instructions. Unless you are implementing maskable interrupts, you can set the **E** and **I** bits to zero. Similarly, the **T**race and **P** bits can be set to zero unless you implement tracing. See the notes at the end of the handout on the instruction set for more information about these flags

Adding or subtracting two n-bit numbers can require an n+ 1 bit number to fully express the result. When the result requires more bits than are available, we have an overflow or carry condition. Overflow occurs for two's complement numbers under the conditions indicated in Table 1.

Operation	Operand A	Operand B	Result
A+B	>= 0	>= 0	< 0
A+B	< 0	< 0	>= 0
A-B	>= 0	< 0	< 0
A-B	< 0	>= 0	>= 0

Table 1. Overflow Conditions

An easy way to determine overflow is to exclusive-OR the carry-in of the high-order bit with the carry-out of the high-order bit. It is left as an exercise for the student to verify that this detects overflow. Carry occurs when the result of adding unsigned n-bit numbers exceeds 2^n -1, or the result of subtracting unsigned numbers is negative. The only way to determine Zero is to check that all of the bits are zero with an OR or OR-like function. Consider what an overflow means to your zero check. The Negative bit can be set from the high-order bit (again, consider what to do in the case of an overflow). The L bit is set assuming unsigned numbers, i.e., that all 16 bits of the operand indicate values, rather than sign, or, that all numbers are positive. The compare instruction does a subtraction of the value in ***src1*** register from that in the ***src2*** register. It is not possible to get an overflow when numbers of the same sign are subtracted. In the compare instruction, operands are treated as both signed and unsigned integers at the same time; the processor does not know which they are. The programmer does know what type of data the operands represent, so can choose the correct condition on which to conditionally branch or jump.

Be sure to clear the flags when they should be cleared (as well as setting them at the appropriate times). Do not change flags when they should not be changed. This can be implemented by clocking the flip-flop for a particular flag bit whenever an instruction, which is to set or clear the bit has been executed. You could include instructions for loading and storing the processor status register (LPR and SPR). These instructions would move data from one of the general-purpose registers into the status register, or read the status register into a general-purpose register. These instructions are not included in the baseline machine; the baseline machine requires the PSR to be modified only by arithmetic instructions and reset. If you are not implementing the LPR and SPR instructions, you can save area by just implementing the flag flip-flops, which you need (5 for the baseline processor).

Implementation of Flags

The flags for addition are straightforward. Let c_n be the carry out of the most significant bit and c_{n-1} be the carry into the most significant position. Then $C = c_n$, and $F = c_n$ XOR c_{n-1}. The most common way of doing subtraction when two's complement numbers are used is to add the one's complement of the number to be subtracted and make the carry in to the least significant bit equal to one. In this case the carry flag is given by $C = c_n_bar$, and F is the same as for addition. In the compare operation (CMP, CMPI) $L = c_n_bar$, and
$N = a_{n-1}$ XOR b_{n-1} XOR $c_n_bar = a_{n-1}$XOR b_{n-1}_bar XOR $c_n = s_{n-1}$ XOR c_n XOR c_{n-1}
where n-1 refers to the most significant bits in computing A-B, and s_{n-1} is the sum bit. Some of these flags change if a "true" subtraction is done in SUB and CMP.