CAD4 The ALU Fall 2009

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 of the instructions above 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, etc.), which set the processor's flags for conditional branches, can be implemented by performing a subtraction and obtaining 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 may need a NOR tree to verify that no result bit is a 1. Note that some fast adder designs may give you the zero detection "for free" or at least provide some of it. When subtracting, Carry_in = 1, so the only way to get a 0 out is if all of the bits are propagating (which means Carry_out will be 1).

There are two general approaches in designing the ALU. The first one is conceptually simpler but the second one may result in a more efficient implementation.

1. Separate Logic Blocks

Your ALU must perform six different functions; this could be accomplished as shown below, with the adder unit performing three of these, and simple logic blocks performing the other three. Then a four-to-one multiplexer (or tri-state buffers) could be used to select the correct function.

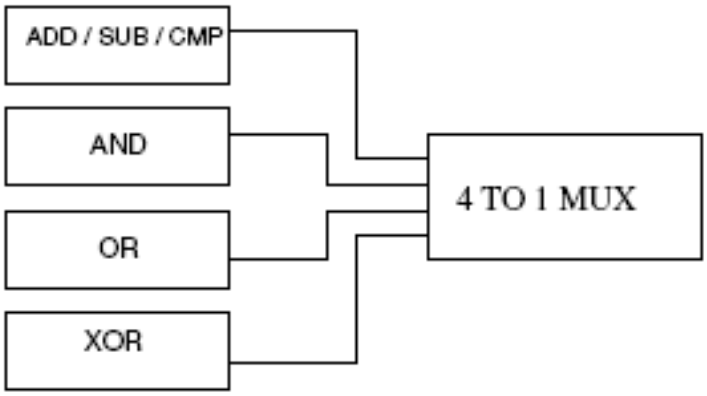

Figure 1: Simple ALU approach.

2. Modified Ripple Carry Adder

An alternative is to share as much of the logic as possible and even embed some logic into the mux. For example, if P is "XOR" and G is "AND", you may compute those and use them in the adder. Then XOR = P, AND = G, OR = P OR G, and ADD = P XOR Carry_in. Since Carry_in will be the latest to arrive you could mux between P, G, P OR G (think about how a 2-to-1 NAND mux could mux between these three things with appropriate control signals) and XOR with Carry_in AND-ed with the "ADD" control signal. There are many approaches, in fact, once you allow the logic to mix together. Another one which is a terrible idea (since it adds logic to the carry path) but has always been presented with CAD4 is on the next page and is included to get you thinking about what is possible.

Sum: S = ABC + AB'C' + A'B'C + A'BC'
Carry: Cout = AB + BC + AC

If we define

Half sum: H = AB' + A'B

then we could write the adder equations as

Sum: S = HC' + H'C
Carry: Cout = AB + HC

From the equations it is clear that when C is held at logical 0, the sum output is an XOR of A and B.

S = AB' + A'B (XOR operation)

Also, when C is held at logical 0, then,

Cout = AB (AND operation)

When C is logic 1, then

Cout = AB + A + B = A + B (OR operation)

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.

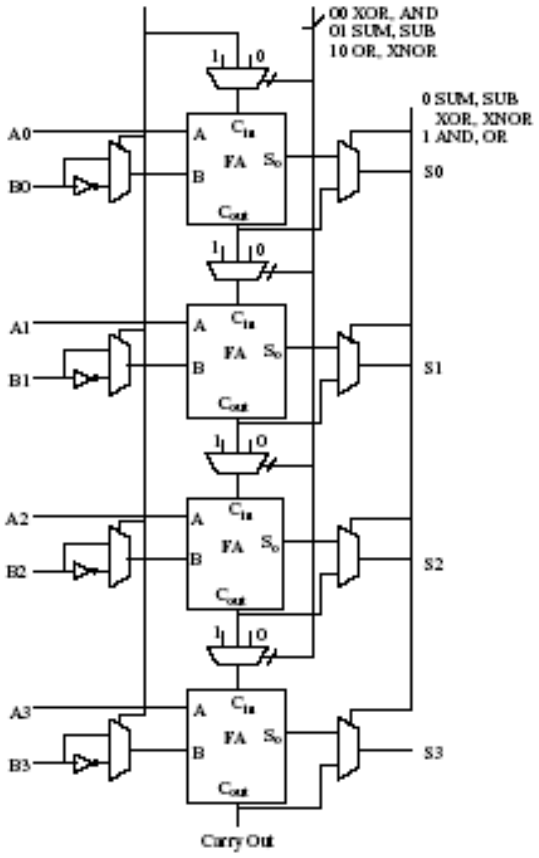

Figure 2: Merged arithmetic/Logic functions.

3. Adder Designs

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 could then be cascaded to form an adder of desired width.

The truth table of a 1-bit adder is as follows:

C	A	B	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Hence, we may write the standard adder equation in the form:

Sum S = ABC + AB'C' + A'B'C + A'BC'

Carry Cout = AB + BC + AC = AB + C(A+B)

We now discuss two different kinds of full adders.

1. Transmission Gate Adder

Fig. 3 is the schematic of a transmission gate adder. The XOR function is implemented on the left. Its outputs are then used to implement the sum and the carry. It has 4 transmission gates, 4 inverters and an XOR gate. Its advantages include equal SUM and CARRY delay times, and non-inverted SUM and CARRY output signals.

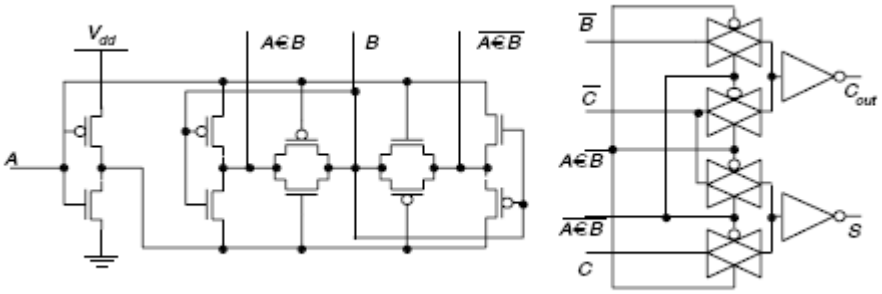

Figure 3: Transmission Gate Adder.

2. Combinational Adder

One example of a combinational adder is shown in Fig. 4. Of course, there are many ways to implement the adder equations, depending upon which gates one chooses to use. In this example, the CARRY signal is used in the generation of SUM (so SUM will be delayed with respect to CARRY). It has 24 transistors, the same as the transmission gate adder.

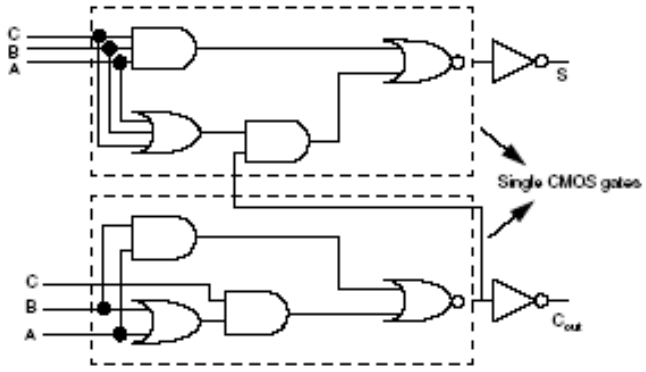

Figure 4: Combinational Adder.

Procedure

Your ALU must be pitch/bit-slice width matched to your register file. In this assignment, you are not required to implement the logic to set the 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?). In particular, you must provide a few outputs capable of setting all of the flags for the controller. Sending the entire output to the controller to detect a 0 is not acceptable.

1. Schematic Design

The ALU is the key block of your datapath and can be the majority of delay in the execute cycle. (If you have a very fast adder, then the memory is likely to be the bottleneck) This handout assumes a ripple-carry adder based ALU with a few different choices for the full-adder design. You may also choose to implement any of the carry lookahead schemes discussed in lecture (Carry-bypass, basic CLA, carry-select, logarithmic lookahead/Brent-Kung, Kogge-stone). These will be less regular and will require significantly more layout and timing effort. Once you choose a particular adder, optimize it for speed. Turning in a design with all minimum-sized devices amounts to a simple logic switch design and will result in significant deductions.

You can implement a normal ripple-carry adder with complex gates, as shown in Fig. 5. 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 closest to the output. This will reduce the body effect on carry-stage transistors.

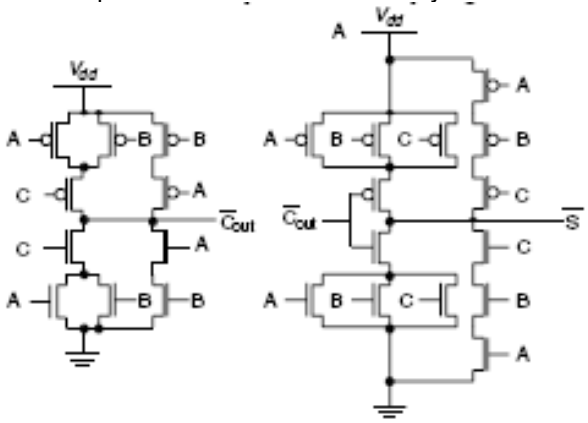

Figure 5: Complex-Gate Combinational Adder.

Remember that ADD, SUB and CMP (immediate versions, too) affect PSR bits. Your ALU should provide easy access to signals necessary for setting these flags. The actual PSR may be in the control unit. Note that zero-detect logic should be included in the ALU so as to avoid routing all 16 sum bits to the control unit. For other flags, just route a few of the higher order sum and carry bits out for later use in the control unit (see appendix for more info). You will also need a sign-extend module to convert immediate data to 16-bit data for the ALU.

2. Logic Verification
Use NCVerilog to verify each of the six functions (i.e. reg-reg and reg-imm versions). Test your zero-detect logic as well.

3. Layout
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 and setting processor status bits. 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.

Be sure to run the buses over your cells. Bit-slice width matching with other datapath modules is very important. Save vertical routing resources for datapath integration. In the layout of a fast adder, you may find this particularly difficult to do. If you find it necessary to use a lot of vertical routing resources within the leaf cell, you could opt to place the ALU on the top or bottom of the datapath such that fewer signals must run over the ALU.

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 (if displacement calculations are done in the ALU) or from the instruction register (immediates). A little thought about floorplanning of your datapath organization in advance will help a lot in getting a good final layout.

4. Design Verification
Do DRC and LVS to verify the layout. Make sure that the DRC "gridcheck" rules are clean for the top level of the ALU hierarchy (you can ignore violations in the lower levels). Save the DRC and LVS reports in your CAD4 directory.

5. Analog Simulation
Extract the parasitic capacitances. Use HSpice to get the worst case delays for the 1-bit cell and for the full 16-bit ALU. Note that you should simulate the full 16-bit ALU if you choose to implement any of the fast adders (Brent-Kung, Carry Select) and that with the fast adders the critical path may not be obvious.

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). NCVerilog traces should show that the 16-bit ALU works for all six (more if you implement more) functions with a few tests per function (Make separate .ps files for each function so that bus values are readable in the printout). It's fine to implement carry-lookahead schemes but note that they will be less regular and will require more layout work.
- 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 will 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 and any sub-circuits you use.
- NCVerilog .ps files for the 16-bit ALU. These should demonstrate all ALU functions (multiple files).
- HSpice .png files showing the longest rise and fall delays for each output of the 1-bit ALU or full 16-bit ALU. 1-bit ALU simulations are fine for ripple carry implementations but you must do 16-bit simulations for anything else. Explain the delays, i.e., which path was critical in determining the delays. For ripple carry implementations calculate the worst case 16-bit delay from the 1-bit delays.
- Clean DRC report (except for lower level "gridcheck" rules).
- Clean LVS report.
- A README explaining the choice of the adder style and floorplanning, along with the names/descriptions of the various files you're submitting. Feel free to include any other pertinent comments (e.g., placement of PSR in the control or the datapath).

Deadline

You need to turn in CAD 4 by **Wednesday, Oct. 14th, 2009, 7pm**.

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 Over**F**low 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. 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.

Table 2: Overflow Conditions.

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

Carry occurs when the result of adding unsigned N-bit numbers exceeds 2^N-1, or the result of subtracting unsigned numbers is negative.

Because zero detection only occurs while performing a subtraction there are actually two ways to go about it. The obvious way to implement the zero detection is to check that all of the bits are zero with an OR or OR-like function. This can be done with a pseudo-nMOS style gate, a tree, or a ripple. The other way is to AND all of the P bits, assuming that P is implemented as an XOR. In the logarithmic lookahead designs this is inherently done for you. In other fast adder designs the base of the tree is already done and you just have to connect the blocks. The Negative bit can be set from the high-order bit. 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 (Rdest in the ISA handout) from that in the *src2* register (Rsrc in the ISA handout). 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 type they are. The programmer, however, does know what type of data the operands represent, and 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 that is allowed 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 Cn be the carry out of the most significant bit and Cn-1 be the carry into the most significant position. Then C = Cn, and F = Cn XOR Cn-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 = !Cn, and F is the same as for addition. In the compare operation (CMP, CMPI) L = !Cn, and N can be derived in several ways:

$$N = a_{n-1} \oplus b_{n-1} \oplus \overline{c_n} = a_{n-1} \oplus \overline{b_{n-1}} \oplus c_n = s_{n-1} \oplus c_n \oplus c_{n-1}$$

where n-1 refers to the most significant bits in computing A-B, and Sn-1 is the sum bit.