ALU CONTROLLER

PROJECT DOCUMENTATION

ALU DESIGN

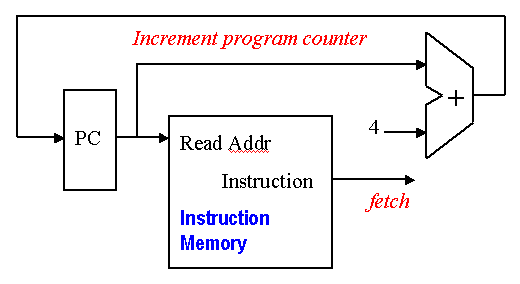
Logisim

The ALU is a component that performs one of a number of arithmetic operations on two 32-bit inputs.

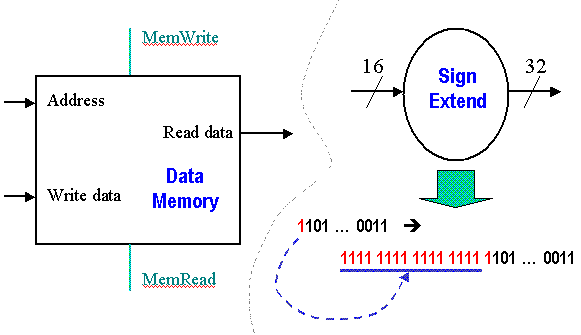
1. Explain the process of developing your alu.circ. How did you get started? What is the sequence of steps you took to complete it? (3-5 sentences)

The datapath is the "brawn" of a processor, since it implements the fetch-decode-execute cycle. The general discipline for datapath design is to (1) determine the instruction classes and formats in the ISA, (2) design datapath components and interconnections for each instruction class or format, and (3) compose the datapath segments designed in Step 2) to yield a composite datapath.

Simple datapath components include *memory* (stores the current instruction), *PC* or program counter (stores the address of current instruction), and *ALU* (executes current instruction). The interconnection of these simple components to form a basic datapath is illustrated in Figure 4.5. Note that the register file is written to by the output of the ALU. As in Section 4.1, the register file shown in Figure 4.6 is clocked by the RegWrite signal.

  
  
**Figure 4.5.** Schematic high-level diagram of MIPS datapath from an implementational perspective, adapted from [Maf01].

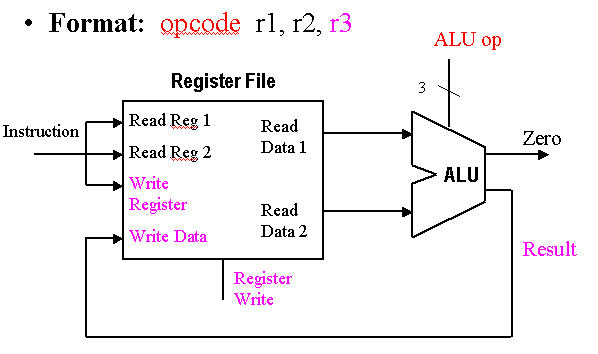
Implementation of the datapath for I- and J-format instructions requires two more components - a *data memory* and a *sign extender*, illustrated in Figure 4.6. The data memory stores ALU results and operands, including instructions, and has two enabling inputs (MemWrite and MemRead) that cannot both be active (have a logical high value) at the same time. The data memory accepts an address and either accepts data (WriteData port if MemWrite is enabled) or outputs data (ReadData port if MemRead is enabled), at the indicated address. The sign extender adds 16 leading digits to a 16-bit word with most significant bit *b*, to product a 32-bit word. In particular, the additional 16 digits have the same value as *b*, thus implementing sign extension in twos complement representation.

  
  
**Figure 4.6.** Schematic diagram of Data Memory and Sign Extender, adapted from [Maf01].

#### 4.2.1. R-format Datapath

2. Explain the process of developing your alu-control.circ. How did you get started? What is the sequence of steps you took to complete it? (3-5 sentences)

Implementation of the datapath for R-format instructions is fairly straightforward - the register file and the ALU are all that is required. The ALU accepts its input from the DataRead ports of the register file, and the register file is written to by the ALUresult output of the ALU, in combination with the RegWrite signal.

  
  
**Figure 4.7.** Schematic diagram R-format instruction datapath, adapted from [Maf01].

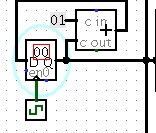
3. Explain the difference between overflow and carry for two’s complement adding and subtracting. (2-4 sentences)

Overflow and carry out are philosophically the same thing. Both indicate that the answer does not fit in the space available. The difference is that carry out applies when you have somewhere else to put it, while overflow is when you do not. As an example, imagine a four bit computer using unsigned binary for addition. If you try to add 10102+111210102+1112 without the word length restriction, you get 100012100012 The high bit does not fit in our word. If this word is all the space allocated to this variable, it represents overflow as the result is too large to represent. If we have a two word space allocated to this variable, it becomes a carry out and is stored in the higher word. We would then represent the addition as 0000 10102+0000 01112=0001 000120000 10102+0000 01112=0001 00012 and the carry is explicit.

4. Explain how your implementation of alu-control.circ works. What approaches did you use to implement all the different cases? (3-6 sentences)

Approach

. The three memories contain the test inputs. You can use the poke tool  to set the Test # register



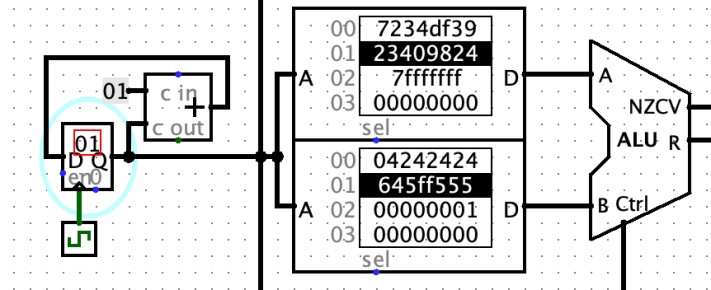
to the Test # given in the error message. Our test’s error message said

Test # ALUFlags Result ...

1 8 87a08d79

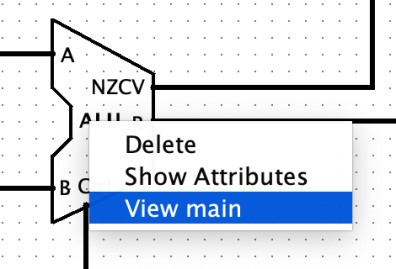
1 9 87a08d79

The Test # is 1, so we would want to set the Test # register to 1 to see which inputs for A, B, and ALUControl were being tested.



In this case it was A=0x23409824, B=0x645ff555, ALUControl=7 (that is, add). Indeed, this addition should lead to an overflow, but our ALU is outputting that overflow=0.

From here, to see why our ALU is outputting the wrong answer, right click the ALU and choose view Main.



This will take you into your alu.circ. The test inputs will be coming in so that you can investigate what is happening on all the wires. **Trace wires backwards from output towards the input.** That is, start at the ALUFlags output, and look for where oVerflow is coming from. Keep tracing backwards until you find an unexpected value on a wire; that will help you find the cause.

• Tip: To see the value on a wire either:

o Use the poke tool (hand with pointer finger) o Or place a Probe, found under Wiring folder

## Description

The inputs and outputs of the ALU are as follows.

Input

|  |  |  |
| --- | --- | --- |
| **Name** | **Bit width** | **Description** |
| A | 32 | First operand |
| B | 32 | Second operand |
| ALUControl | 4 | the operation the ALU should compute to obtain the value of output Result |

Output

|  |  |  |
| --- | --- | --- |
| **Name** | **Bit width** | **Description** |
| ALUFlags | 4 | {Negative, Zero, Carry, oVerflow} |
| Result | 32 | Result of the operation |

Here are the individual bits in the ALUFlags

|  |  |  |
| --- | --- | --- |
| **Name** | **Bit width** | **Description** |
| oVerflow | 1 | 1 iff operation is add or sub and there was signed overflow |
| Carry | 1 | 1 iff operation is add or sub and there was carry out |
| Zero | 1 | 1 iff Result is equal to 0 |
| Negative | 1 | 1 iff Result is negative |

Here is the specification for the Result.

|  |  |  |
| --- | --- | --- |
| **ALUControl** | **ALU operation name**  **(*NOT* an ARM instruction)** | **Result** |
| 0 | and | A AND B |
| 1 | or | A OR B |
| 2 | xor | A XOR B |
| 3 | not | NOT A |
| 4 | lsl | B << A |
| 5 | lsr | B >> A (logical) |
| 6 | asr | B >> A  (arithmetic) |
| 7 | add | A+B |
| 8 | sub | A-B |

RESULTS

v2.0 raw

20010001

00210820

00210820

00210820

00210820

00210820

AC010000

8C020000

00211824

AC630000

8C640000

AC64000