Design of the Register Files & Integration with the ALU Datapath

CS/ECE 3710

Robert Christensen

James Lewis

Levi Balling

September 27, 2011

## Design of the ALU

|  |  |  |
| --- | --- | --- |
| **opcode name** | **opcode** | **opext** |
| ADD | 0000 | 0101 |
| ADDI | 0101 | XXXX |
| ADDU | 0000 | 0110 |
| ADDUI | 0110 | XXXX |
| ADDC | 0000 | 0111 |
| ADDCU | 1010 | 0101 |
| ADDCUI | 1010 | 0110 |
| ADDCI | 0111 | xxxx |
| SUB | 0000 | 1001 |
| SUBI | 1001 | xxxx |
| CMP | 0000 | 1011 |
| CMPI | 1011 | xxxx |
| CMPU/I | 1010 | 0010 |
| AND | 0000 | 0001 |
| OR | 0000 | 0010 |
| XOR | 0000 | 0011 |
| NOT | 1010 | 0011 |
| LSH | 1000 | 0100 |
| LSHI | 1000 | xxxx |
| RSH | 0000 | 1110 |
| RSHI | 1110 | xxxx |
| ALSH | 1010 | 0001 |
| ARSH | 1010 | 0100 |
| nop | 0000 | 0000 |
| MOV | 0000 | 1101 |
| MOVI | 1101 | xxxx |

The design and implementation of the ALU closely followed the guidelines and suggestions from course material. All operations which were requested to be implemented in the ALU lab handout were successfully implemented, tested, and simulated.

The ALU operations that were implemented are shown in Table 1. This includes addition, subtraction, comparison, bit logical operations, shift, and move instructions. The opcodes assigned to each instruction were taken from the ISA handout. If the ISA handout did not include the particular instruction an opcode was selected which would not conflict with any opcodes currently assigned to the other instructions.

All ALU instructions that were planned were implemented. No plans have been made to implement multiplication or division. All additional instructions which will be implemented for the design of the computer are non-ALU instructions.

The ALU takes as input two 16 bit vectors A and B, a 4-bit opcode and a 4-bit opcode extension to describe what operation will be performed by the ALU. Also, a carry bit is taken as input from the processor status register. The carry input flag is only used for the ADD with carry instructions.

The ALU has two separate outputs. A 16 bit vector S, which is the result of performing the desired operation on the A and B inputs. A status flag is also emitted which signals certain conditions as a byproduct of performing the desired operation.

Table - ALU instructions with opcode

The status flags contain only 5 bits. The purpose of each bit is described in

|  |  |  |
| --- | --- | --- |
| **CLFZN[4:0]** | **description** | **summary** |
| CLFZN[4] | carry | set when a carry is generated on an ADD instruction |
| CLFZN[3] | low flag | set in CMP instructions when (unsigned)B<(unsigned)A |
| CLFZN[2] | flag bit | signals overflow in ADD and SUB instructions |
| CLFZN[1] | zero bit | set in CMP instruction when A==B |
| CLFZN[0] | negative bit | set in CMP instructions when (signed)B < (signed)A |

. The meaning of each bit has come from the CR16A Programmer's Reference Manual. All ADD and SUB instructions have the potential to set the flag bit. Only CMP instructions will set the low flag, zero bit, and negative bit values of the status flags. This was done for consistency. The CR16A Programmer’s Reference Manual describes that only one CMP instruction is needed. In order to be consistent with this manual and implement the requested instructions every CMP instruction is not different from any of the other CMP instructions implemented in this ALU.

The code for the ALU consisted of a series of casex statements, one for each possible operation implemented. The body of the statement set the appropriate registers as described in the CR16A Programmer's Reference Manual. Many operations performed identical operations, such as ADD and ADDI, as well as every CMP instruction. The control logic will be the part of the processor that will make the distinction between ADD and ADDI by doing the sign extend on the immediate value. Some operations do actually perform the exactly the same operation such as the CMP which was described above. This was done because of the inability to implement fall though in the Verilog language.

Table - Processor Status Register

The synthesis report for the ALU module showed that the delay for this circuit is 14.951ns, with 12 levels of logic. The path of the longest delay in the circuit is from opcode<3> to S<6>. Table 3 shows area utilization of the synthesized model. It is seen from this table that 261 4 input LUTs are being used.

|  |  |  |
| --- | --- | --- |
| **CLFZN[4:0]** | **description** | **summary** |
| CLFZN[4] | carry | set when a carry is generated on an ADD instruction |
| CLFZN[3] | low flag | set in CMP instructions when (unsigned)B<(unsigned)A |
| CLFZN[2] | flag bit | signals overflow in ADD and SUB instructions |
| CLFZN[1] | zero bit | set in CMP instruction when A==B |
| CLFZN[0] | negative bit | set in CMP instructions when (signed)B < (signed)A |

Table - Area utilization of synthesized ALU

|  |  |  |  |
| --- | --- | --- | --- |
| **Logic Utilization** | **Used** | **Available** | **Utilization** |
| Number of Slices | 145 | 4656 | 3% |
| Number of 4 input LUTs | 261 | 9312 | 2% |

## Testing of the ALU

The ALU was tested using a sequence of coded tests. Each instruction that needed to be tested was fed predetermined input and the output was tested with the predetermined desired result of the operation. In addition several random values were generated as input to the ALU and the output was verified to be correct.

Also, the Post-Synthesis Simulation Model was compared with the Pre-Synthesis Model. A large number of random inputs were generated and fed into both models. The outputs of both were compared for equality with each other. No differences were found in this comparison. This showed that the synthesized model performed as expected from the description of the ALU provided.

## Future Plans for the ALU

The ALU is a basic unit of the Computer system being developed. All operations which were initially planned for the ALU were implemented. Some operations, however, will require some additional operations to be performed by another component before or after the ALU.

The difference between adding two values stored in registers and adding a register value to an immediate value is not noticeable to the ALU. When this is encountered the control logic will place the value of the register on one input to the ALU and do a sign extend on the immediate value and place that on the other input to the ALU. After this happens, the ALU will compute the result of the operation. This functionality will be implemented when the control logic is designed.

In the next section the design of the Register file will be explained. After the register file has been properly connected to the ALU many operations of the computer will be operational.

## Design of the Register File

The register file was designed using a MUX based design. The register file was designed in such a way that it could communicate with the ALU as described in Figure 1.

The register file takes as input two 4-bit inputs that indicate what registers are selected, a 16-bit register value with a 4-bit input to select what register to write the 16-bit value to. A write enable flag is taken as input to signal if the 16-bit input value should be written to the register file. Also, a clock is taken as input to signal when data should be latched.

When given the appropriate input output of the register file are the two 16-bit register values that were selected.

Between the register file and the ALU a mux is present which selects whether the input to the ALU will be from the register file or a sign extended immediate value. However, this part of the processor logic will be part of the control logic and it has not been implemented, so the mux is only selecting values originating from the register file.

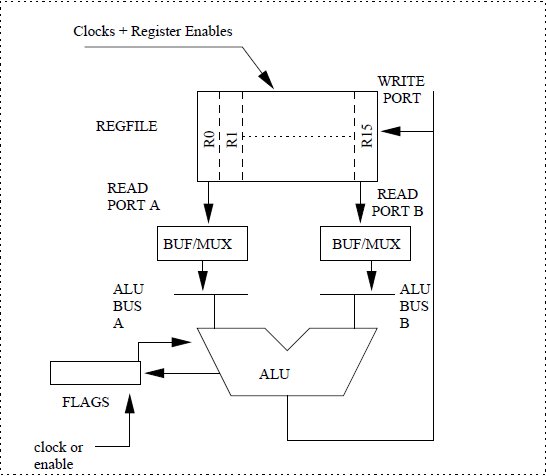


Figure - register find with ALU

## Testing of the Register File

A simple test bench was written to test the register file. This test bench would calculate each element of a fibonacci sequence from to where the fibonacci number is in the register. The test bench would check that the values stored in the registers were calculated and stored correctly.

The register file simulated with not errors or warnings. Table 4 shows the area utilization of the synthesized register file. This module used no BUFs, but did use 256 flip flops to store the data in each of the 16 registers. This is exactly as expected because there are 16 registers and each register contains 16 bits of data, so the expected number of flip flops required to store this data is .

The synthesis report showed the longest delay in the register file module has three levels of logic. The reported path starts with the Select Input value to the register value being set. The reported time for this logic is about 5 ns.

Table - Area utilization of synthesized register file

|  |  |  |  |
| --- | --- | --- | --- |
| **Logic Utilization** | **Used** | **Available** | **Utilization** |
| Number of Slices | 275 | 4656 | 5% |
| Number of Slice Flip Flops | 256 | 9312 | 2% |
| Number of 4 input LUTs | 278 | 9312 | 2% |

## Interaction between Register File and ALU

The register file will be a source of ALU operands and be a destination of some ALU operations. The logic of whether or not a register value or an immediate value will be used as input to the ALU will be performed in the control logic. Figure 1 shows the basic interaction between the register file and the ALU with logic performed by the control logic removed.

Our test program consists of a state machine, which runs through 16 states, the first state stores a value from the input switches when a set button is pressed. (SetA stores the value in register 0, SetB stores the value in register 1) The remaining states each set the value of a register to the sum of the contents of the previous registers, to calculate a Fibonacci-like sequence with the given seed values. A clock divider is used so that the results of each step can be seen as it occurs. The last state displays the value stored in the last register. The four DIP switches on the Spartan-3e board are used to set the initial seeds used, the west button is SetA, which sets the value in register 0, the south button is SetB which sets the value in register 1, and the east button is a global reset. The test program was synthesized and tested on the board and functions appropriately.