Lab 2 Report — Register Files and Datapath

Computer Design Laboratory ECE 3710
Fall 2021
The University of Utah

Jacob Peterson
Computer Engineering 2022
University of Utah
Salt Lake City, UT

Brady Hartog
Computer Engineering 2022
University of Utah
Salt Lake City, UT

Isabella Gilman
Computer Engineering 2023
University of Utah
Salt Lake City, UT

Nate Hansen
Computer Engineering 2023
University of Utah
Salt Lake City, UT

***Abstract*—This paper reports on the ALU design and synthesis along with the regfile design and datapath integration for the CompactRISC16 (CR16) processor. First, the Algorithmic Logic Unit (ALU) design and implementation is explored in detail including: port list of the ALU Verilog module, opcodes and their associated encodings, operation status flags and their associated encodings. Then, the regfile design and implementation is discussed thoroughly with the following highlights: behavioral design with a Verilog `generate` block, function detail, port list, and block diagram. Additionally, the overall organization and implementation of the ALU, regfile, and the datapath between the two is discussed in detail. Finally, the synthesis statistics are reported (number of Look-up Tables (LUT), ALMs, FFs, BUFs used) and a timing analysis report is included.**

I. Introduction

The ALU and regfile which this paper covers are critical to the design and implementation of the CR16 processor. In the completed processor, the data path they create will handle all instructions programmed into the machine. In this design, the regfile is responsible for temporarily remembering values that are currently in use, while the ALU handles all arithmetic done by the processor. By combining the two, we create datapath that is able to read values from the regfile, process the values in the ALU, then save the output back in the regfile. This pattern makes up the core functionality of our processor.

II. The CR16 ALU

The CR16 ALU implements the opcodes shown in Table I. Our ALU implementation uses a large case block for the given opcode that executes behavioral operations for the respective operation. The following shows some arithmetic operations that our behavioral Verilog in our ALU uses:

- + is the addition operator which is used to add numbers in both signed and unsigned contexts with 2's complement numbers.
- − is the subtraction operator which is used to add numbers in both signed and unsigned contexts with 2's complement numbers.
- > and < are numerical comparison operators which is used to set various status bits for a given operation.
- `&`, |, ^, and ~ are boolean logic operators for AND, OR, XOR, and NOT respectively which are used for specific opcodes and in combinational logic for setting the flag bits on certain operations.
- << and >> are logic shift operators which perform bit-shift operations on the given operands.
- <<< and >>> are arithmetic shift operators which perform sign-extending bit-shift operations on the given operands.

TABLE I
ALU Opcodes

ALU Opcode	Instruction Encoding	Description
ADD	0	Signed addition
ADDU	1	Unsigned addition
ADDC	2	Signed addition with carry
ADDCU	3	Unsigned addition with carry
SUB	4	Signed subtraction
SUBU	5	Unsigned subtraction
AND	6	Bitwise AND
OR	7	Bitwise OR
XOR	8	Bitwise XOR
NOT	9	Bitwise NOT
LSH	10	Logical left shift
RSH	11	Logical right shift
ALSH	12	Arithmetic (sign-extending) left shift
ARSH	13	Arithmetic (sign-extending) right shift

As seen in Table I, there is no regard for immediate values since the ALU synthesizes as purely combinational logic that is not concerned with the immediate values since those are going to be decoded from a given processor instruction and passed to the ALU individually. The module signature containing the port list is seen in Figure 1.

III. The CR16 Regfile

The CR16 register file consists of 16 general-purpose, 16-bit registers. The block diagram is shown in Figure 2.

In our implementation, the register file module, `cr16_regfile`, uses the `parameter` construct to parameterize: 1) the bit width of the registers, and 2) the number of registers. These parameters are both 16 by default. The module has a global RESET control which is tied to every register; it has a multi-channel ENABLE control

TABLE II
ALU STATUS BIT MAPPINGS

Status Bit	Status One-Hot Encoding	Description
CARRY	`5'b00001`	MSB carry out for unsigned addition and subtraction
LOW	`5'b00010`	`B < A` for unsigned subtraction
FLAG	`5'b00100`	MSB carry out for signed addition
ZERO	`5'b01000`	Set when `C == 0`
NEGATIVE	`5'b10000`	`B < A` for signed addition and subtraction

```
module cr16_alu
      #(parameter integer P_WIDTH = 16)
       (input wire I_ENABLE,
        input wire [3 : 0] I_OPCODE,
        input wire [P_WIDTH - 1 : 0] I_A,
        input wire [P_WIDTH - 1 : 0] I_B,
        output reg [P_WIDTH - 1 : 0] O_C,
        output reg [4 : 0] O_STATUS);
```

Fig. 1. The CR16 ALU module signature with port list.

that can enable the registers individually. In practice, this ENABLE control enables only one register at a time, which is the register to be written on any given instruction. The value to be written is input via the register bus, which is connected from the output of the ALU to every register. The output of the register file is a vector of size (number of registers) × (bit width) of all data in the registers.

The output vector is connected to the read ports of the ALU. In keeping with modern convention, the read ports are designed as multiplexors rather than as tri-state buffers. For the register file, this means that, unlike with tri-state buffers, both read ports have one register selected at any given time. This design choice presents a challenge for instructions that wish to use the same register as both an operand and the result; such instructions cannot be executed in a single clock cycle, as this

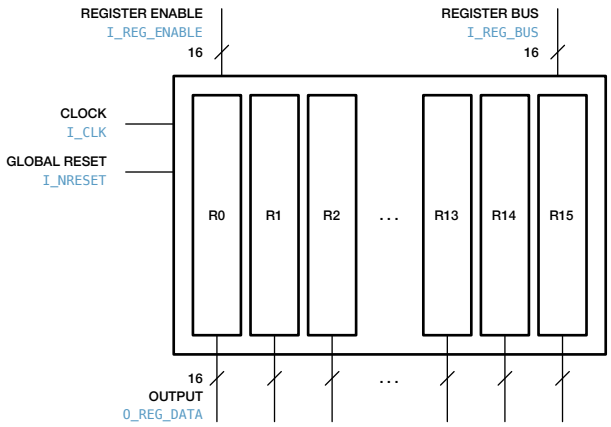

Fig. 2. Block diagram of the CR16 register file.

approach would require the register to be simultaneously read by a read port and written by the register bus. The solution to this challenge is the subject of future work.

IV. DATAPATH

The datapath of the cr16 architecture is a union of the ALU to the register file. We made a few design decisions here to aid in incremental implementation and testing of the unit. The following is a block diagram detailing the top-level view of our datapath, including a dummy finite state machine with hard-coded instructions to aid in the testing and synthesis:

A. Inputs

The inputs of the datapath will be controlled later on by the CPU controller. They consist of the following:

1) I_NRESET: A global reset for the register file and flags register.
2) I_OPCODE: The opcode to be supplied to the ALU.
3) I_ENABLE: Global enable signal for ALU and read from all registers.
4) I_CLK: Clock signal for sequential logic in registers.
5) I_REG_A_SELECT/I_REG_B_SELECT: Each of these are 4-bit selector signals designed to pass one register from each write port (A and B) to the next element on the datapath. We'll discuss this further when we elaborate on multiplexer implementation.
6) I_IMMEDIATE: 16-bit value to be supplied as an immediate if the instruction implements one.
7) I_IMMEDIATE_SEL: Selector for the multiplexer that decides whether to allow an immediate value or the value from read port A to be active at the input A of the ALU.

B. Outputs

The outputs are much more simple. They are mainly used for testing and will likely be removed when SRAM and CPU control are implemented:

1) O_RESULT_BUS: The bus that connects the output of the ALU to the write port of the register file. This output is necessary to properly test the functionality of the datapath.
2) O_STATUS_FLAGS: The bus that connects the ALU flags output to the write port of the flags register. This output is necessary to ensure flags are reported and written correctly.

We decided to implement multiplexer-based control flow since that is used most commonly in modern CPU architectures. Selector signals are supplied to the read ports A and B, and to the multiplexer array at ALU input A which decides whether to pass the Regfile read port A or the immediate value supplied to the datapath.

C. Read Port Multiplexing

The multiplexer logic at read ports A and B is quite extensive. The register file gives access to a 2-dimensional array containing all the data stored in each of the registers. To

consolidate this to two selected ports, we first implemented a model for a 16-to-1 multiplexer. This multiplexer consists of a 16-bit input, a 1-bit output, and a 4-bit selector as shown in Fig. 3.

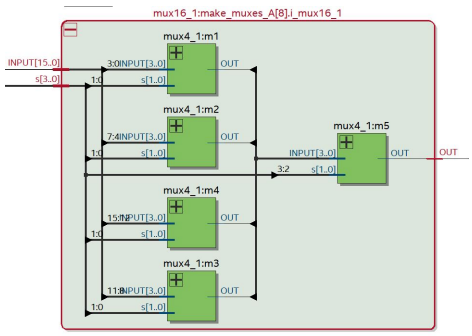

Fig. 3. 16-to-1 Multiplexer with 4-bit selector as seen in Quartus RTL viewer.

To establish one read port, we instantiate 16 of these 16-to-1 multiplexers and connect each one to the nth bit of every register. For example, the first 16-to-1 mux is connected to the LSB of each r0-r15. This allows us to feed the same 4-bit selector to each mux and draw out a 16-bit vector consisting of the data from one of the registers in the Regfile. We specify *which* register by setting the selector signal to the decimal value 0-15 corresponding to the desired register. The connections appear as shown in Fig. 4.

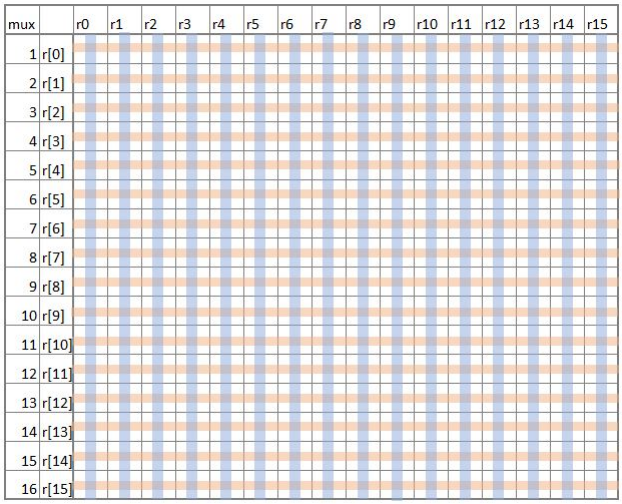

Fig. 4. Read Port to 16x16-to-1 Mux Connections

D. Immediate Value Multiplexing

To control the use of immediate values, we can use a vector of 2-to-1 multiplexers–essentially a 32-to-16 mux–to select either the immediate supplied to the datapath or the value from read port A. Each mux is supplied the same 1-bit selector signal from the CPU control. This prevents conflicts on bus A. A diagram of this implementation is shown in the full block diagram in Fig. 5.

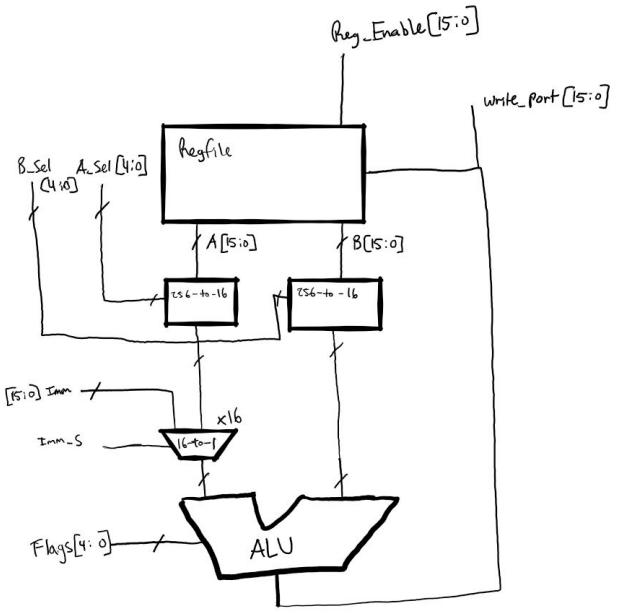

Fig. 5. Block diagram for datapath. Global enables and resets omitted for graphic clarity.

This datapath will synthesize sequential logic, and therefore it is easiest to test it for correct functionality using a Finite State Machine.

V. Testing with Testbenches and the FSM

We conducted testing in two ways: by simulating a dummy set of instructions in a testbench, and by hard-coding some CPU control into a simple FSM to synthesize with the datapath and display outputs on the FPGA.

A. The Testbench

The testbench we have written consists of four separate tests designed to exercise multiple unique functions of the datapath.

1) Test 1 simulates 15 iterations of the Fibonacci Sequence by writing a 1 into r0 and r1, then writing each subsequent answer into a subsequent register through r15.
2) Test 2 loads a 0 value into all 16 registers and then attempts to subtract 1 from register r1 to show signed arithmetic is possible.
3) Test 3 loads 0111 into r1 and 0100 into r2. Various logical Boolean operations are conducted on these values to determine the correct function of logical operations.
4) Test 4 loads integer 1 into registers r1 and r2. The test then attempts and writes a total of 15 sequential bit-shifts on registers r1-r15 to ensure proper functionality of the shift operation.

The first three tests in the testbench show working functionality of the datapath, while the fourth is a work-in-progress.

B. The FSM Hardware Test

We have hard-coded an FSM program using case statements that will execute 7 stages of the Fibonacci sequence and populate the first 8 registers of the register file. The demonstration

of this functionality is available to view at https://youtu.be/-uqqciOsI4k.

The demo shows the use of a single FPGA button as a global reset. We also use a button from the FPGA to simulate the advancement of each state in the FSM program. This button is tied to the clock cycle so that the datapath and Regfile respond to the changes in the input values each time that the state advances. This is a simple demonstration that will extend into larger functionality as we move forward to the implementation to the full FSM of the CPU control.

VI. Synthesis Reports and Timing Analysis

We reviewed the synthesis reports of the datapath module independent of the FSM and top-level module we used for hardware testing. The following data was obtained.

A. Synthesis Reports

Analysis & Synthesis Summary	
Analysis & Synthesis Status	Successful - Fri Sep 24 20:38:40 2021
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	CPU_16-bit_Proj
Top-level Entity Name	cr16_datapath
Family	Cyclone V
Logic utilization (in ALMs)	N/A
Total registers	261
Total pins	69
Total virtual pins	0
Total block memory bits	0
Total DSP Blocks	0
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0
Total DLLs	0

Fig. 6. Summary data from the Quartus map synthesis report after synthesizing ”cr16_datapath.v” as the top-level module.

As is shown in the summary of the synthesis report (Fig. 6), the datapath occupies 261 total registers and contains 69 pins. This is substantial. It is likely that the logic surrounding the multiplexing of the regfile’s read ports is occupying memory. Our code is functional in nature, so Quartus will make decisions behind the scenes that we cannot control regarding the synthesis of multiplexer logic. This is made even more apparent from the perspective of the netlist and resource occupation, which can be seen in Fig. 7 and Fig. 8. There is a high volume of fanout on each 16-bit bus that has to propagate signals through 16 inputs of a mux, and our design utilizes a total of 32 16-to-1 multiplexers. The implementation of these muxes is carried out hierarchically beginning from the level of a 2-to-1 mux, so the resulting logic circuit will contain many levels of hierarchical multiplexing. This increases the LUT depth significantly. The synthesis report details the average LUT depth at 8.82, while the maximum LUT depth is 10. This is considerable, but understandable considering our design strategy. This could be avoided in multiple ways. Perhaps there is a more efficient design strategy that would allow us to use less hardware to properly multiplex the read ports of the register file while still taking advantage of the portability of multiplexers. Further, we considered implementing the control of data reading using tri-state buffers. We were advised not to do this in the specification, but we believe it would decrease the hardware complexity and reduce LUT depth.

Post-Synthesis Netlist Statistics for Top Partition	
Type	Count
arriav_ff	261
ENA SCLR	256
SCLR	5
arriav_lcell_comb	508
arith	67
0 data inputs	2
1 data inputs	33
4 data inputs	32
extend	7
7 data inputs	7
normal	434
2 data inputs	25
3 data inputs	30
4 data inputs	51
5 data inputs	61
6 data inputs	267
boundary_port	69
Max LUT depth	10.00
Average LUT depth	8.82

Fig. 7. Netlist data from the Quartus map synthesis report after synthesizing ”cr16_datapath.v” as the top-level module.

Considering the average fan-out of our design is also a crucial element of efficiency analysis. While we do not have any other implementations of this particular piece of hardware to which we could compare our design, it is instructive to review the data. Total fan-out in our implementation is reported to be 3653, which is considerable. Average fan-out for each net is 4.08. The maximum fanout node is the global reset input, which is connected to each register in the Regfile and the flags register. The fanout of this node is so high (277) because each 16-bit register is likely synthesized using an array of 1-bit flip-flops. This means that the I_NRESET signal will be connected to 16 flip-flops * 16 registers, or about 256 ports. The other 21 connections exist within the Regfile and the flags register. Other fanout occurs like this among connections like I_ENABLE that have to be connected to many single-bit elements of the low level hardware synthesis. Overall it’s

Analysis & Synthesis Resource Usage Summary	
Resource	Usage
Estimate of Logic utilization (ALMs needed)	405
Combinational ALUT usage for logic	508
-- 7 input functions	7
-- 6 input functions	267
-- 5 input functions	61
-- 4 input functions	83
-- <=3 input functions	90
Dedicated logic registers	261
I/O pins	69
Total DSP Blocks	0
Maximum fan-out node	I_NRESET~input
Maximum fan-out	277
Total fan-out	3653
Average fan-out	4.03

Fig. 8. Resource data from the Quartus map synthesis report after synthesizing ”cr16_datapath.v” as the top-level module.

hard to claim that our design is more or less efficient than others. It occupies comparatively few of the 85k available Logic Elements, so there are no constraints that would force us to rework our implementation for lack of resources.

B. Timing Analysis

We conducted timing analysis using the Quartus Timing Analyzer tool. This analysis contains the maximum time delay data from the input ports to the output ports of the datapath module, independent of the testing FSM and top-level module. We have included the top 25 longest propagation paths in Fig. 9. Regarding both Rise-Rise and Fall-Fall time delays,

Propagation Delay

	Input Port	Output Port	RR	RF	FR	FF
1	I_IMMEDIATE[2]	O_RESULT_BUS[3]	17.009	16.141	16.590	18.925
2	I_IMMEDIATE[2]	O_RESULT_BUS[15]	16.918	17.325	17.682	18.630
3	I_REG_B_SELECT[3]	O_RESULT_BUS[15]	17.180	17.565	18.023	18.408
4	I_REG_B_SELECT[2]	O_RESULT_BUS[15]	16.993	17.400	17.986	18.393
5	I_REG_B_SELECT[2]	O_RESULT_BUS[6]	16.738	17.303	17.731	18.296
6	I_IMMEDIATE[2]	O_RESULT_BUS[9]	16.728	16.232	16.655	18.285
7	I_REG_B_SELECT[2]	O_RESULT_BUS[2]	16.322	17.237	17.315	18.238
8	I_IMMEDIATE[2]	O_RESULT_BUS[2]	16.414	16.334	16.638	18.202
9	I_IMMEDIATE[2]	O_RESULT_BUS[14]	16.652	16.846	17.423	18.172
10	I_REG_B_SELECT[0]	O_RESULT_BUS[15]	16.792	17.199	17.700	18.107
11	I_REG_B_SELECT[2]	O_RESULT_BUS[3]	16.408	17.079	17.401	18.080
12	I_IMMEDIATE[2]	O_RESULT_BUS[11]	16.281	16.471	16.885	18.060
13	I_REG_B_SELECT[3]	O_RESULT_BUS[6]	16.692	17.257	17.401	18.024
14	I_REG_B_SELECT[0]	O_RESULT_BUS[6]	16.537	17.102	17.445	18.010
15	I_IMMEDIATE[2]	O_RESULT_BUS[6]	16.666	17.231	17.427	17.992
16	I_REG_B_SELECT[2]	O_RESULT_BUS[14]	16.734	16.986	17.727	17.987
17	I_IMMEDIATE[6]	O_RESULT_BUS[15]	16.975	17.360	17.601	17.986
18	I_IMMEDIATE[2]	O_RESULT_BUS[13]	16.453	16.847	17.224	17.958
19	I_REG_B_SELECT[3]	O_RESULT_BUS[14]	16.934	17.109	17.777	17.952
20	I_REG_B_SELECT[3]	O_RESULT_BUS[2]	16.234	17.088	17.077	17.939
21	I_REG_B_SELECT[3]	O_RESULT_BUS[13]	16.735	17.081	17.578	17.924
22	I_REG_B_SELECT[2]	O_RESULT_BUS[12]	16.517	16.925	17.510	17.918
23	I_REG_B_SELECT[2]	O_RESULT_BUS[13]	16.535	16.922	17.528	17.915
24	I_REG_B_SELECT[3]	O_RESULT_BUS[12]	16.717	17.037	17.560	17.880
25	I_REG_A_SELECT[1]	O_RESULT_BUS[15]	16.648	17.055	17.401	17.808
26	I_IMMEDIATE[2]	O_RESULT_BUS[5]	16.213	16.155	16.523	17.794
27	I_IMMEDIATE[2]	O_RESULT_BUS[12]	16.435	16.850	17.206	17.787
28	I_REG_B_SELECT[3]	O_RESULT_BUS[4]	16.259	16.931	17.110	17.782

Fig. 9. Timing analysis of top 25 maximum propagation times along the datapath from all inputs to all outputs.

it is apparent that the longest propagation times occur along the signal path from the read port selectors to the write port along the result bus. This makes sense since there are 4 levels of multiplexing occurring in the read ports, and the signal propagation is delayed in that hierarchy. Since the register values are not seen as primary inputs, they will not appear in this analysis, but they have essentially the same critical path through each level of the multiplexer tree. It is interesting to note that there doesn't seem to be a measurable difference between the propagation time of the MSB of selector B and the LSB of selector B. We would expect the selector bit connected to the top level of muxes to take longer to propagate than the bit connected to the final stage, but this is not the case. It is also not obvious why the selector signal for read port A propagates faster than that of read port B, since both seem to have symmetrical critical paths. The signal providing an immediate value to the datapath also seems to show long propagation delays. Since it is only sent through one level of multiplexers to the ALU, we would expect it to propagate faster than the selector signals. It would seem to have a shorter critical path. However, the timing analyzer shows that the propagation of immediate values is more delayed than we expected. All of these delay times rest at or near 18ns-19ns. This will likely be too great a delay to utilize the kinds of clocks that modern processors can use, but perhaps a slower clock like the ones on the FPGA will not run too fast for our implementation.

VII. Conclusion

The datapath implementation we have built is subject to modification as the project progresses and we better understand the overall functionality of the CR16 CPU. We have yet to implement a proper system that will load immediate values into the Regfile. The opcode mapping will likely look different from the perspective of the CPU control and FSM, so when we implement instruction decoding we will have two separate instruction encodings that will belong to the ALU and the CPU respectively. The CPU will have to interpret instructions exactly as they are decoded from the ISA, and this will allow the CPU to handle immediate instructions separately and control this behavior within the datapath and Regfile. We intend to clarify these details as we move forward so we can integrate a better version of this datapath/ALU module into our final working implementation of a CR16 CPU.