Logan Johnson
ECE 2544
Dr. Cooper
17 November 2021

Learning Experience F2 Report

Objectives

The purpose of this assignment is to implement a 16-bit function unit consisting of 13 different operations into a preexisting simple computer model. Then design an efficient opcode to control the operation of the computer.

Design approach

The register transfer system consists of 2 major components; the function unit and the transfer system. The function unit performs a desired operation on 2 imputed 8-bit vectors. The function unit is separated into three general function types; arithmetic (or addition) operations, logic operations, and shift operations.

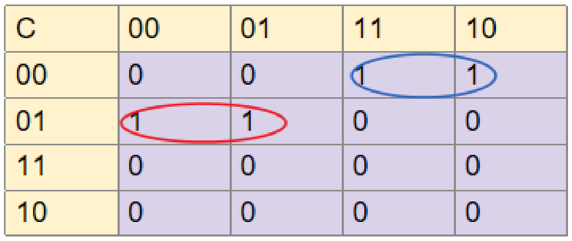
**light green boxes are the addition operations, mid-green boxes are the logic operations and dark green boxes are the shift operations*

SW[5:2]	Operation
0000	A+B
0001	A
0010	A-B
0011	-B
0100	B-A
0101	-A
0110	NAND
0111	NOR
1000	A’
1001	mod8
1010	DEV4
1011	CIR-R
1100	CIR-L

The arithmetic section of the circuit uses 8 one-bit full adder circuits to complete these arithmetic operations. This full adder circuit is loaded by a mux for each operand. These muxes decipher when each vector (A or B) should be added (Ex. if the operation is A + B, the A mux will input the A vector and the B mux will input the B vector, or for the operation movA the A mux would input A and the B mux would input a zero vector). I also designed a logic circuit to determine when the first bit of the adder should have a carry-in bit. This occurs whenever there is a negative number (Ex. A-B and -A).

SW[3]	SW[2]	SW[1]	SW[0]	C in
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Cin = (~SW[3] & SW[2] & ~SW[1]) | (~SW[3] & ~SW[2] & SW[1])

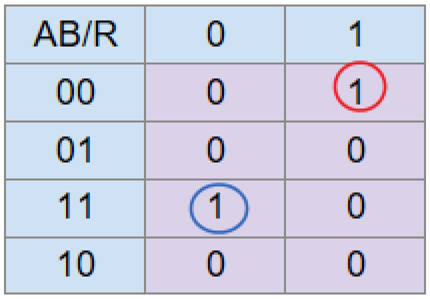


C	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	0	0	0	0
10	0	0	0	0

The addition circuit is also responsible for the status bits V-overflow and C-carryout. The carry-out bit comes directly from the most significant bits full adder. The overflow bit is found by creating a logic circuit using the most significant bit of both the inputs and the output. Unfortunately, the function was not able to be simplified using a k-map.

A[7]	B[7]	Result[7	V
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

V = (~A & ~B &R) | (A & B & ~R)



AB/R	0	1
00	0	1
01	0	0
11	1	0
10	0	0

The logic section of the function unit consists of a mux that performs the desired logic operation on the A and B vectors.

The shift section is made of a mux that only affects the B operand. The mux changes the position of each of the imputed bits to “shift” the vector. The mux is also responsible for inputting a zero for a bit if the operation calls for it.

The function unit module calls the addition module, the logic module, and the shift module, and OR’s their outputs together to produce one 8-bit result. The status bits Z-zero and N-negative are also defined in the function unit module. The Z is created by combining all the result bits together in a NOR gate. The N status bit is just the most significant bit of the result.

In order to convert my 8-bit function unit to a 16-bit function unit, I had to multiply the number of each type of mux by two and double the number of full adders used. This was possible because my modules created for my muxes were only one bit, meaning I was able to just double the total number used. The only section that required special attention was the shift section because the old shift locations didn't compensate for the new 16-bit data. I first planned out what each bit output should be based on the function code, and then I implemented the modified locations into the pre-existing mux.

Logic Circuits

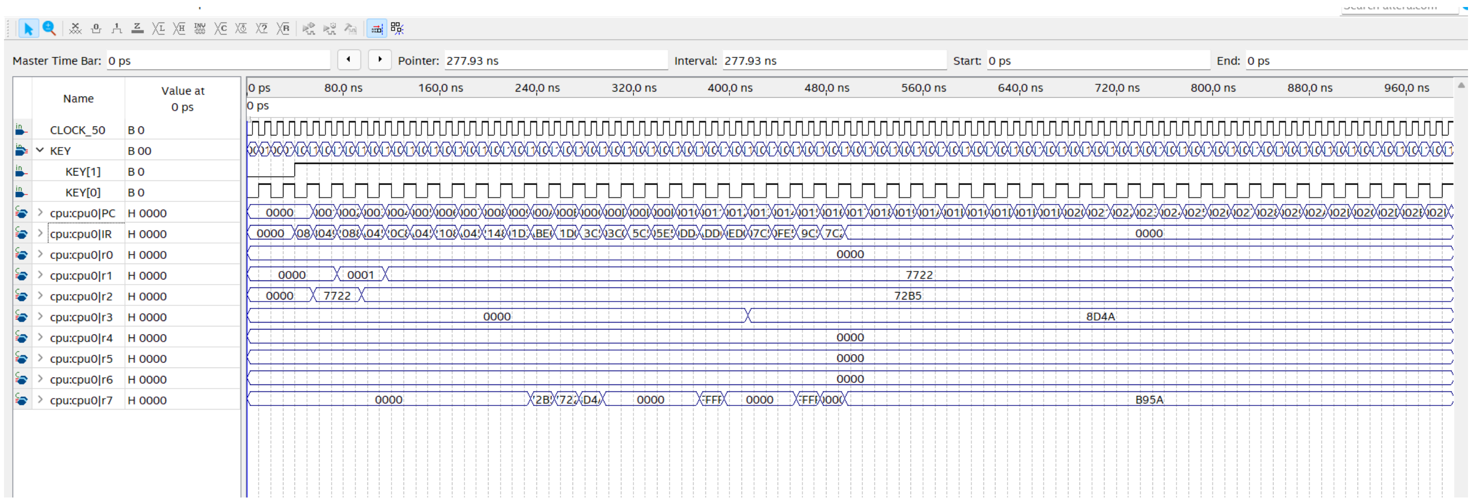
My circuit generally consisted of the Addition circuit, and logic circuit, shift circuit. Most of these circuits were able to be created using one or more muxes. The logic sedition and the shift section were both created using one mux and the addition and transfer circuit were created using 2 muxes. My approach for scaling the muxes to 16-bits was to create a module for a 1-bit mux and then scale it up to 16-bits in another module. This helped me simplify the overall code and make it easier to implement. When creating these muxes, I decided to use simple assign statements with AND and OR gates to make the transistor and propagation delay easier and more precise. I also decided to organize my modules by the section of code they were a part of to increase the overall organization of the code.

Opcode design

When designing my opcode, I first listed out the operations and their corresponding FS code. From here I mapped what control word values correspond to each operation. I used the book's instruction decoder to determine which control word corresponds to which digit of the opcode. From here I was able to determine the correct opcode by lining up what the control words should be for each operation with the opcode that would make that happen. In the final design the control word assignments where (the number represents the digit of the 16-bit input code with 15 being the most significant bit) MB=15, FS[3]=12, FS[2]=11, FS[1]=10, FS[0]=9 & (14 & 15)’, MD=13, RW=14’, MW=14 & 15’, PL=14 & 15, JB=13, and BC=9.

Operation	Mnemonic	Opcode	MB	FS[3:0]	MD	RW	MW	PL	JB	BC
A+B	add	0000000	0	0000	0	1	0	0	0	0
A	Move A	0000001	0	0001	0	1	0	0	0	1
A-B	SubAB	0000010	0	0010	0	1	0	0	0	0
-B	NegB	0000011	0	0011	0	1	0	0	0	1
B-A	SubBA	0000100	0	0100	0	1	0	0	0	0
-A	NegA	0000101	0	0101	0	1	0	0	0	1
NAND	NAND	0000110	0	0110	0	1	0	0	0	0
NOR	NOR	0000111	0	0111	0	1	0	0	0	1
A’	NOTA	0001000	0	1000	0	1	0	0	0	0
MOD8	MOD8	0001001	0	1001	0	1	0	0	0	1
DEV4	DEV4	0001010	0	1010	0	1	0	0	0	0
CirR	CSR	0001011	0	1011	0	1	0	0	0	1
CirL	CSL	0001100	0	1100	0	1	0	0	0	0
Load	LD	0010000	0	0000	1	1	0	0	1	0
Store	ST	0100000	0	0000	0	1	1	0	0	0
Jump	JMP	1110000	1	0000	0	0	0	1	1	0
Immediate add	ADI	1010000	1	0000	0	1	0	0	0	0
Immediate subtraction	SUBI	1010010	1	0010	0	1	0	0	0	0
Immediate NAND	NANDI	1010110	1	0110	0	1	0	0	0	0

Simulation


Master Time Bar: 0 ps Pointer: 277.93 ns Interval: 277.93 ns Start: 0 ps End: 0 ps

Name	Value at 0 ps
CLOCK_50	B 0
KEY	B 00
KEY[1]	B 0
KEY[0]	B 0
cpu:cpu0|PC	H 0000
cpu:cpu0|IR	H 0000
cpu:cpu0|r0	H 0000
cpu:cpu0|r1	H 0000
cpu:cpu0|r2	H 0000
cpu:cpu0|r3	H 0000
cpu:cpu0|r4	H 0000
cpu:cpu0|r5	H 0000
cpu:cpu0|r6	H 0000
cpu:cpu0|r7	H 0000

Evaluation of implementation

My implementation was overall successful proven by the simulation values seen above. I was able to see the exact values that I expected out of each portion of the simulation. An example of this is the first operation. The operation load (meaning take the data stored in a register in the data memory and store it in a register in the register file) from register one to register 2. My simulation shows the data from the first spot in the data.txt file being implemented into the second register in the register file. This is exactly what I expected the result to look like.

My implementation was very efficient with most of my control words being implemented without the use of any logic gates, and when a gate was needed I was able to implement it having at most 2 inputs. This efficiency lowers the overall transistor count of my circuit and minimizes the propagation delay seen by my control unit.

Transistor Count

*I assumed that gates grouped together (Ex. (a&b&c)) made one gate. I also assumed that every time a not was assigned, it counted as a new not gate.

function circuit

	NOT	2AND	3AND	5AND	2OR	3OR	4OR	5OR	2NAND	2NOR	Total transistors
Full adder	6	3	4	X	X	1	1	X	X	X	80
Mux-a	17	X	X	5	X	X	X	1	X	X	106
Mux-b	14	X	X	4	X	X	1	X	X	X	86
lmux	7	X	X	3	X	1	X	X	1	1	66
muxs	7	X	X	4	X	X	1	X	X	X	72
arith+	2	1	2	X	2	X	X	X	X	1	42
main+	X	X	X	X	X	8	X	X	X	16	80

Full Circuit

	Full adder	Mux-a	Mux-b	lmux	muxs	arith+	main+
arith	8	8	8	X	X	1	X
log	X	X	X	8	X	X	X
shift	X	X	X	X	8	X	X
Full design	X	X	X	X	X	X	1
Total Transistors	X	X	X	X	X	X	3402

The total transistor count for the function unit is 3402 transistors.

The instruction decoder consisted of 3 NOT gates and 4 2 input AND gates making the total transistor count for the 30 transistors.

Propagation Delay

Function unit

	NOT	2AND	3AND	5AND	2OR	3OR	4OR	5OR	2NAND	2NOR	Total Delay (ps)
Full adder-S	1	X	1	X	X	X	1	X	X	X	250

Full adder-C	X	1	X	X	X	1	X	X	X	X	175
Mux-a	1	X	X	1	X	X	X	1	X	X	325
Mux-b	1	X	X	1	X	X	1	X	X	X	300
lmux	X	X	X	1	X	1	X	X	X	1	300
muxs	X	X	X	1	X	X	1	X	X	X	275
arith+c	1	7	1	X	X	7	1	X	X	X	1400
arith+v	X	1	X	X	1	X	X	X	X	X	150
main+	X	X	X	X	X	1	X	X	X	16	100
main+n	X	X	X	X	X	X	X	X	X	X	0
main+z	X	X	X	X	X	X	X	X	X	16	200

The critical path for each major component

*FA is full adder

	FA-S	FA-C	mux-a	mux-b	lmux	muxs	arth+ c	arth+ v	main +	main +n	main +z	Total delay (ps)
arith-c	X	X	X	X	X	X	1	X	X	X	X	1400
arith-v	X	X	X	X	X	X	X	1	X	X	X	150
arith-s	1	7	X	X	X	X	X	X	X	X	X	1400
log	X	X	X	X	1	X	X	X	X	X	X	300
shift	X	X	X	X	X	1	X	X	X	X	X	275
main+n	X	X	X	X	X	X	X	X	X	0	X	0
main+z	1	7	X	X	X	X	X	X	X	X	1	1600

The critical path consists of the 1 counter, 1full adder, 1 Num0, 1 Extra0, and 1 Code making the total circuits delay 1650ps.

The critical path for the instruction decoder consists of 2 2-input AND gates and one NOT gate making the total delay 175ps.

Conclusion

My circuit was able to successfully complete this assigned objective. The function unit was able to complete my desired 16-bit operations and the instruction decoder successfully deciphered my designed opcode. The simple computer was able to function in the way I intended it by reading or writing both registers in the register file or registers in the data memory. Overall my circuit was efficient, I was able to implement circuit simplification techniques such as k-mapping to limit the total transistor count. I also made my opcode efficient making the number of transistors and the critical path of my instruction decoder as small as possible. Even though I implemented techniques to simplify the circuit, I also made the trade-off of having higher readability over the absolute minimum transistor count. This is because I designed my Verilog expressions using assign statements that modeled a circuit's simplified function. While this made it much easier for the reader to understand, the transistor count and propagation delay suffered slightly due to repeated logic gates (like multiple not's of a single input). I believe this tradeoff was worth it because it made my code much easier to read and troubleshoot. Looking at the big picture, this project introduced me to both the control unit and the function unit of the simple computer I will be fully designing in the future.