Mustafa Acıkaraoğlu
Merve Barın

COMP303 – Project Report

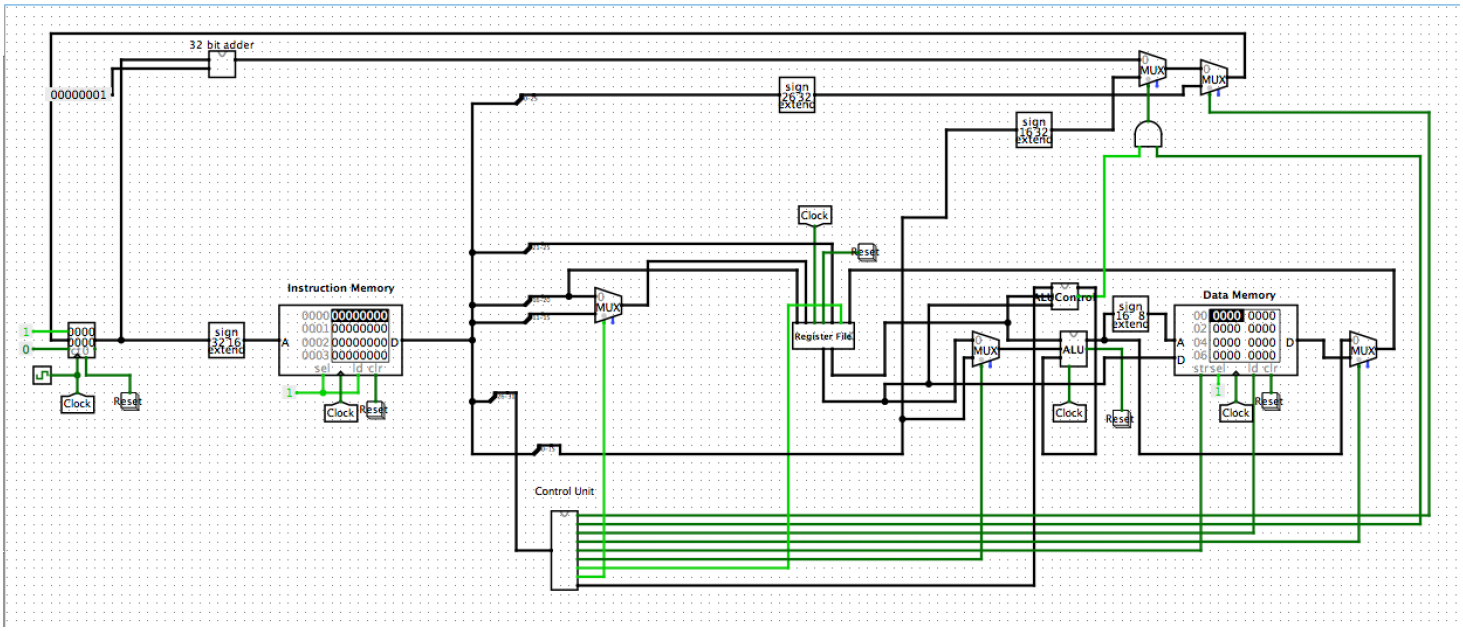


Figure 1: Circuit Design on Logisim

In this Project, we were asked to design a 16-bit single-cycle processor using Logisim, however our processor must support 32-bit instructions even though the processor was 16-bits. The most important sub-circuits were the control unit and the ALU control unit. The regular control unit generates correct signals for multiplexers for certain types of instructions. ALU control unit manages the instructions that are not completely arithmetic, for example lw, sw, beq, blez etc. Our register file consisted out of 8 16-bit registers. The ALU of our processor has a special ability for multiplication instruction, since it was asked that the “mult” "instruction must store the high and low bits into special registers that are write protected, so we have created additional 2 registers to track the multiplication result. Also for our instruction we have created a 32-bit adder that keeps track of the carry result of the previous add instruction. Our special instruction is called “addhi” and it became handy when we tried to multiply 16-bit negative numbers and add them. By our “addhi” instruction we can correctly calculate results that are bigger than 16-bits and store them in 2 separate registers, because we need the keep track of the lower carry bit and sum the high bits with that carry in mind. Down below you can find our instructions and their corresponding control signals.

Instr.	Opcode	Type	Jump	Branch	MemRead	MemToReg	MemWrite	ALUSrc	RegWrite	RegDest
Add	100000	R	0	0	0	0	0	0	1	1
Sub	100010	R	0	0	0	0	0	0	1	1
Mult	011000	R	0	0	0	0	0	0	0	0
And	100100	R	0	0	0	0	0	0	1	1
Or	100101	R	0	0	0	0	0	0	1	1
Addi	001000	I	0	0	0	0	0	1	1	0
Sll	000000	R	0	0	0	0	0	0	1	1
Slt	101010	R	0	0	0	0	0	0	1	1
Mfhi	010000	R	0	0	0	0	0	0	1	1
Mflo	010010	R	0	0	0	0	0	0	1	1
Lw	100011	I	0	0	1	1	0	1	1	0
Sw	101011	I	0	0	0	1	1	1	1	0
Beq	000100	I	0	1	0	0	0	0	0	0
blez	000110	I	0	1	0	0	0	0	0	0
j	000010	J	1	0	0	0	0	0	0	0
addhi	111111	R	0	0	0	0	0	0	1	1

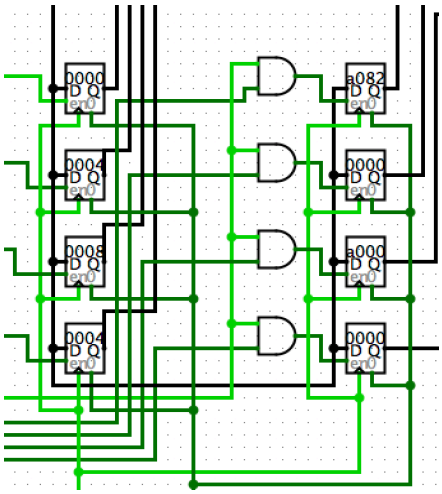
Table 2: Summary of Control Signals

As it can be seen from the Control Signals table, our instruction is a R-type instruction. As an additional feature, we have written an assembler for our processor in Java. This Java code generates the correct hexadecimal machine code for instructions. Just run the shell script provided to generate the machine code (shell script compiles runs and deletes the .class file automatically). For testing the program we have written the given C code in our own instructions and generated the corresponding machine code.

```
addi $1,$0,0 # 1'st address counter, initial counter
addi $2,$0,4 # 2'nd address counter
addi $3,$0,4 # threshold for the counter
addi $4,$0,0 # sum of lower bits
addi $5,$0,0 # sum of higher bits
beq $1,$3,16 # check if $1 is equal to $3 if not cntinue to loop ,else exit by jumping to 16'th line which is empty
lw $6,$1,0 # load from memory (1'st array) = [MEM + $1 + 0000 0000 0000 0000]
lw $7,$2,0 # load from memory (2'nd array) = [MEM + $2 + 0000 0000 0000 0000]
mult $6,$7 # multiply $6 and $
mflo $6 # load low bits of the result
mfhi $7 #load high bits of the result
add $4,$4,$6 # sum the low bits (tracks the carry as well)
addhi $5,$5,$7 # sum the high bits with addition of the carry as well
addi $1,$1,1 # increase counter, 1'st array address by 1 (normally this is 4)
addi $2,$2,1 # increase 2'nd array address by 1 (normally this is 4)
j 5 (jump to the 5'th line to continue the iteration)
```

Here we can see the assembly code for the provided C code. When this code is run on our processor it generates the same output with the C code and the proof can be seen below.

```
Mustafas-MacBook-Pro:COMP303 mustafa$ cat provided_c_code.c
#include <stdio.h>
void main()
{
int a[4]={5, 7, -2, 40}; /* Initialize vector a */
int b[4]={65, -23, 17, 1024}; /* Initialize vector b */
int i = 0, sum = 0;
for(i = 0; i < 4; i++)
{
sum = sum + a[i] * b[i];
}
printf("%d\n",sum);
}
Mustafas-MacBook-Pro:COMP303 mustafa$ gcc provided_c_code.c -o output
provided_c_code.c:2:1: warning: return type of 'main' is not 'int' [-W
void main()
^
provided_c_code.c:2:1:       change return type to 'int'
void main()
^~~~
int
1 warning generated.
Mustafas-MacBook-Pro:COMP303 mustafa$ ./output
41090
Mustafas-MacBook-Pro:COMP303 mustafa$
```

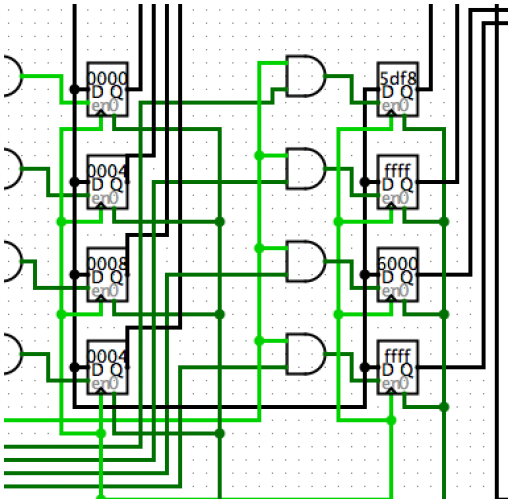


Provided C code output: 41090

Our output: $= (A082)_{16} = (41090)_{10}$

Keep in mind that for the provided instructions we have 2 sum registers. Register $4 is for the lower bits of the result and $5 is for the higher bits of the result, so the whole answer would be $\left((0000)_{higher}(A082)_{lower}\right)_{16} = (41090)_{10}$ which is still correct. We can test the correctness of the functionality of our processor by tweaking the given C code.

```
Mustafas-MacBook-Pro:COMP303 mustafa$ cat provided_c_code.c
#include <stdio.h>
void main()
{
int a[4]={-5, 7, -2, -40}; /* Initialize vector a */
int b[4]={65, -23, 17, 1024}; /* Initialize vector b */
int i = 0, sum = 0;
for(i = 0; i < 4; i++)
{
sum = sum + a[i] * b[i];
}
printf("%d\n",sum);
}
Mustafas-MacBook-Pro:COMP303 mustafa$ gcc provided_c_code.c -o output
provided_c_code.c:2:1: warning: return type of 'main' is not 'int' [-W
void main()
^
provided_c_code.c:2:1:       change return type to 'int'
void main()
^~~~
int
1 warning generated.
Mustafas-MacBook-Pro:COMP303 mustafa$ ./outpur
-bash: ./outpur: No such file or directory
Mustafas-MacBook-Pro:COMP303 mustafa$ ./output
-41480
Mustafas-MacBook-Pro:COMP303 mustafa$
```



Tweaked C code output: -41480

Our Tweaked Output:

$\left((FFFF)_{higher}(5DF8)_{lower}\right)_{16} = (-41480)_2$

As it can be seen from the outputs, our processor is capable of doing even 32-bit operations by using 16-bit registers. Multiplication was handled by keeping the following formula in mind: if the last bit of the binary number is 1 then the number is negative and 2's compliment is taken when multiplied.

$if\ (\ XOR\ (\ A\ B\ )\ ==\ 0; then; Output_{normal}; else\ 2's\ compliment(Output)$

Above formula helps us to understand whether the output will be positive or negative.