Computer Architecture Project

Team 39

Youssef Amro 55-10541
Kamal Said 55-1387
Abduallah Ayman 55-1449
Omar Mohamed 55-1449
Aly Basyouni 55-2712

1 Introduction

In this project, we aimed to simulate a fictional processor design and architecture in C. The architecture we chose is called Double McHarvard with cheese circular shifts. It is a Harvard architecture, which splits up the memory into instruction memory and data memory.

2 Methodology

2.1 What is Double McHarvard with Cheese Circular Shifts?

In this package, we have instruction memory size of 1024*16 bits and a data memory size of 2048*8 bits. We also have 66 registers: 64 8 bit general purpose registers, a 16 bit program counter, and a single 8 bit status register. The status register has 5 flags: A carry flag for whenever the result has carry, a overflow flag for whenever the result overflows, a negative flag, which is 1 whenever the result is negative, a sign flag, which shows us the sign we expect for the result (1 being negative), and a zero flag for whenever the result is zero. The processor also pipelines by fetching, decoding, and executing at the same clock cycle whenever possible.

2.2 Parsing

The first step to execute any program was to read the assembly code and turn it into binary 1s and 0s. We achieved this by using the predefined struct FILE, which has multiple methods that relate to opening and reading a file's content. One of these was fopen, which takes the file path and a character to specify what we want to do with the file. Since we are only going to read the file, we set the character to be "r". We then use an fgets to get a single line of assembly. This is done by giving fgets 3 inputs: a char array to store the line, the max size a single line could get, which we set to 256 since no assembly line will be ever greater than that, and finally a pointer to the file itself. Once fgets sees no more lines to be read, it will return NULL.

We then starting splitting up the instruction depending on the spaces using strtok_r. This method takes in a string pointer, a delimiter, and a another string, and returns the string split up until the delimiter is found, storing the rest the string in the third input. During the first split, we used the returned result to get the opcode. This was done by comparing the split string with the possible operations we have, if they are the same, we return the respective opcode for it. If the opcode is not one of the operations, then the result is undefined behaviour. We then split the string again, and take the returned result to find the first operand. If the string contains a R, we increment the pointer to the string by one, then parse the rest of the string into int. If it does not contain a R, we parse the string without incrementing into int. The third and final split to get the second operand is done in the same way we did the second split.

We then shift the opcode by 12, the first operand by 6, concatenate all of them together by ORing them together, and finally put it into our instruction memory. We repeat this process until we reach the end of the text file. Once the end of the text file is reached, we add at the end of the instruction memory the max value of a 16 bit number, or 0x7FFF in hexadecimal as to mark the end of the instructions when executing the program.

2.3 Variables

We have multiple global variables that help our processor to fetch, decode, execute, and pipeline our instructions. In order to simulate a real processor, anything that was defined in subsection 2.1 as 16 bits had the datatype short int, while anything that was 8 bits had the datatype int8_t. Including the registers and memory mentioned in the subsection, we also had a variable called isBranched that flipped to 1 whenever we jumped/branched to another instruction. It was used to handle control hazards. We also defined a struct called instruction, which was used to help in our pipeline. This struct contained 6 variables:

```
short int fullInstruction; // used to store the full instruction when fetching
int8_t opcode; // used to store opcode when decoding
int8_t r1Address; // used to store R1 when decoding
int8_t r2Address; // used to store R2 when decoding
int8_t imm; // used to store immediate when decoding
int isEnabled; // used to control when we can activate a certain stage
               // explained in more detail in pipeline
```

2.4 Fetching

After reading and parsing our assembly code, our processor starts by fetching the instruction from the instruction memory. We did this by initializing our PC to 0, and using it to find which instruction to fetch from the instruction memory. We then stored this instruction in the instruction struct's fullInstruction. Finally, we incremented our PC by 1 so the next time we fetch we get the next instruction. If the instruction we fetched was 0x7FF, then we disabled the fetch by setting the instruction struct's isEnabled to 0.

2.5 Decoding

Once the instruction is fetched, we start splitting it up into possible parts our processor might want. We do this by shifting the bits we want to the least significant bits, then bit masking the least significant bits we want. We do this for the opcode, shifting 12 bits then masking it with 4 bits, the first operand, shifting 6 bits then masking it with 6 bits, the second operand and immediate by masking it with 6 bits directly. We store all of these values inside the instruction struct.

2.6 Execution

Finally, we start executing the instruction. Depending on the opcode, we either use the immediate or the second operand. If we use the second operand, we use it as an unsigned 6 bit int to fetch the register for the operation. If it is immediate, then we treat it as a signed 6 bit integer.

Our processor supports 12 different operations, as seen in the table below. One thing to note is that whenever we jump/branch as seen in instruction BEQZ and JR, we set isBranched to 1 so that later we can empty the pipeline for the instructions after jumping. Another thing to note is that whenever we call LDI, we check if the 6th bit is 1. If it is, it means that whatever value to be stored in the 8bit register is negative, so with bit extend the immediate then store it inside the register.

Name	Mnemonic	Type	Format	Operation
Add	ADD	R	ADD R1 R2	R1 = R1 + R2
Subtract	SUB	R	SUB R1 R2	R1 = R1 - R2
Multiply	MUL	R	MUL R1 R2	R1 = R1 * R2
Load Immediate	LDI	I	LDI R1 IMM	R1 = IMM
Branch if Equal Zero	BEQZ	I	BEQZ R1 IMM	IF(R1 == 0) { PC = PC+1+IMM }
And	AND	R	AND R1 R2	R1 = R1 & R2
Or	OR	R	OR R1 R2	R1 = R1 | R2
Jump Register	JR	R	JR R1 R2	PC = R1 || R2
Shift Left Circular	SLC	I	SLC R1 IMM	R1 = R1 << IMM | R1 >>> 8 - IMM
Shift Right Circular	SRC	I	SRC R1 IMM	R1 = R1 >>> IMM | R1 << 8 - IMM
Load Byte	LB	I	LB R1 ADDRESS	R1 = MEM[ADDRESS]
Store Byte	SB	I	SB R1 ADDRESS	MEM[ADDRESS] = R1

Figure 1: Table of all our supported operations

We also have to change the status register according to the operation itself. The carry flag is checked every ADD instruction, and is set to 1 whenever the addition of the two operands' unsigned values result in the 9th bit being 1. Overflow is checked for the ADD and SUB instructions, and is set to 1 whenever result is different from what we expect. More precisely, if we're performing addition, then the result must have a different sign than the operands for overflow to have occurred. If we're performing subtraction, then the result must have the same sign as the subtrahend for overflow to have occurred. The sign flag is also checked every ADD and SUB instruction, and is set by XORing the negative and overflow flags. Negative and zero flags are checked every ADD, SUB, MUL, AND, OR, SLC, and SRC instruction. The negative is set to one whenever the result is negative, and the zero flag set to one whenever the result is 0.

2.7 Pipeline

The pipeline for this architecture is relatively simple. We used the struct instruction as defined before to declare 3 variables: fetch, decode, and execute. We had the fetched enabled flag 1, while the decode and execute were 0. The pipeline worked by constantly looping through the 3 instruction structs and checking the enabled flags; if they were 1 we would start that specific phase, otherwise we wouldn't. After passing through the 3 phases', we would copy the fetch instruction

onto the decode and the decode instruction onto the execute. This would result in turning on the phase if it was disabled, since copying the instruction means it inherits it's variables too. We would repeat the process of checking the 3 phases then copying instructions until all 3 structs had their flag disabled. This would occur when fetch finds the 0x7FF instruction, disabling itself, and later cascading, disabling the rest of the stages.

In order to handle control hazards, we used the flag isBranched to detect any jumps. If a jump occurred, then after copying the instructions we disabled both the execute and decode, preventing old instructions from starting, while fetch goes ahead and gets the next instruction after branching/jumping.

3 Results

In order to test our processor, we wrote out assembly code on a text file to test out each of our instructions and the status register. The following is our tests and their results.

3.1 Loading Negative Numbers

Listing 1: Loading a negative number into register 0

```
1 LDI R0 -5
```

The first test we did was to attempt to load negative numbers into the register. This first resulted for 59 to be loaded inside the register. This was because the immediate was 6 bits, but the register itself was 8 bits, so we needed to extend the sign bit. After adding two 1s at the start of each number if it was negative, we get the result we expect, which is R0 = -5 as seen in the image below.

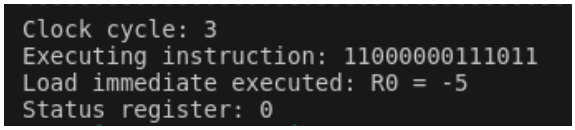


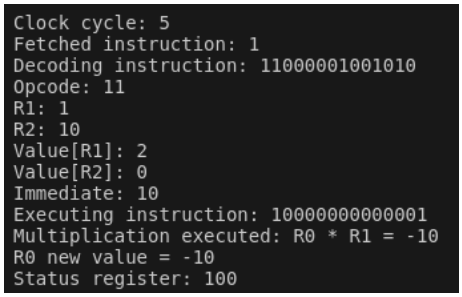
Figure 2: Successful result of loading a negative number into a register

3.2 Arithmetic Instructions

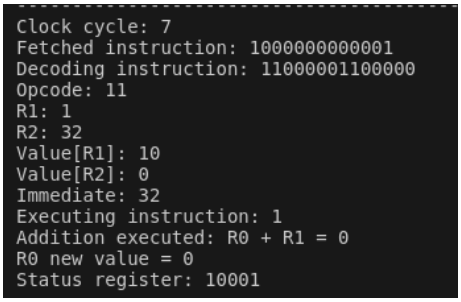
Listing 2: Multiplication, addition, and subtraction

```
1 LDI R0 -5
2 LDI R1 2
3 MUL R0 R1 # should be -10 as shown in figure 2(a)
4 LDI R1 10
5 ADD R0 R1 # should be 0 as shown in figure 2(b)
6 LDI R1 -32
7 SUB R0 R1 # should be 32 as shown in figure 3(a)
8 ADD R0 R1 # should be 0 as shown in figure 3(b)
```

The second test we did included arithmetic instructions. We did not attempt to specifically test overflow or carry here. The results were as expected, and both the negative and zero flag seemed to work correctly. This also showed that our pipeline worked fine. For 8 instructions, the expected amount of clock cycles is $3 + ((8 - 1) * 1)$, or 10, which is exactly the amount of clock cycles our processor took.The below images show the result after each arithmetic operation.

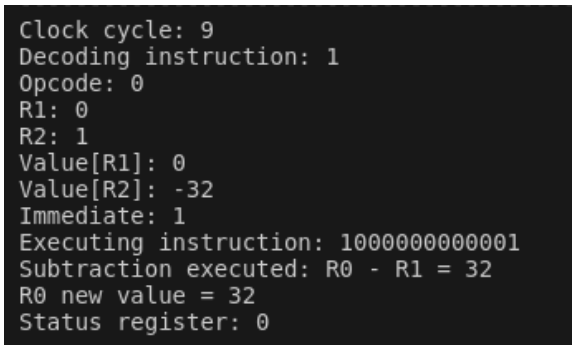


(a) Multiplying R0 and R1

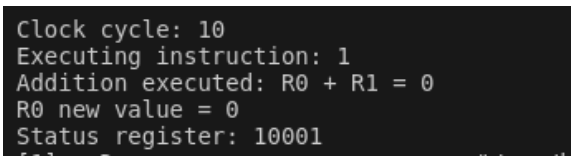


(b) Adding R0 and R1 for the first time

Figure 3: Instructions 3 and 5



(a) Subtracting R0 and R1



(b) Adding R0 and R1 for the second time

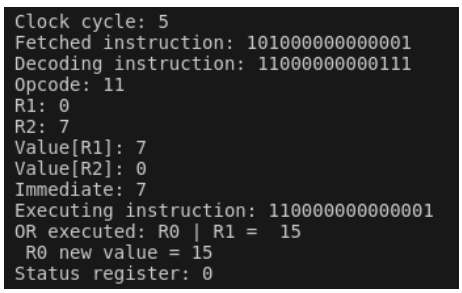
Figure 4: Instructions 7 and 8

3.3 Logical instructions

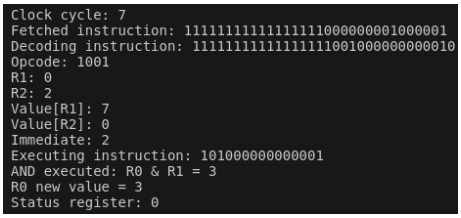
Listing 3: ORing, ANDing, and shifting

```
1 LDI R0 7 # 0111
2 LDI R1 11 # 1011
3 OR R0 R1 # should be 15 (1111) as seen figure 4(a)
4 LDI R0 7 # 0111
5 AND R0 R1 # should be 3 (0011) as seen figure 4(b)
6 SRC R0 2 # should be -64 (1100 0000) as seen figure 5(a)
7 SLC R1 1 # should be 22 (0001 0110) as seen figure 5(a)
```

The third test we did included the logical instructions. This includes ORing, ANDing, and shifting. This test gave us our expected results and clock cycles, taking 9 clock cycles total to finish 7 instructions. Results of each logical operation can be seen below.

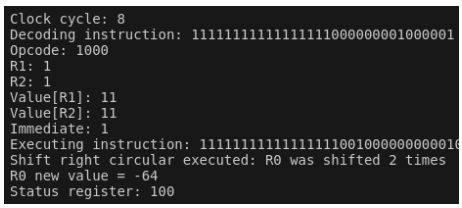


(a) ORing R0 and R1

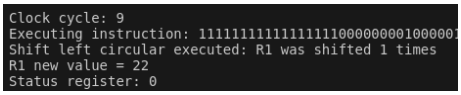


(b) ANDing R0 and R1

Figure 5: Instructions 3 and 5



(a) Shifting right circular R0 by 2



(b) Shifting left circular R1 by 1

Figure 6: Instructions 6 and 7

3.4 Overflow with Carry

Listing 4: Executing arithmetic operations to reach our desired number then adding

```
LDI R1 -32
LDI R2 4
MUL R1 R2 # R1 is now -128
SB R1 0
LB R2 0 # R2 is now -128
ADD R1 R2 # should cause overflow and carry
```

We then started testing the overflow and carry flags. The first was overflow with carry, which occurs when the result has a carry over and needs 1 more bit to fit in the result. The above assembly code does this by having the max negative number that can fit in a signed 8 bit integer (-128) added to itself, giving us a result of -256, which cannot fit in 8 bits, causing overflow and carry. The above code also shows us that the sign flag is working, as we expect negative but the number itself is 0. The execution of the final add instruction can be seen below.

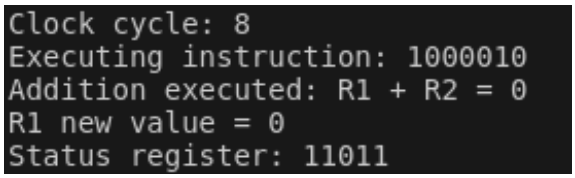


Figure 7: Both carry and overflow flag are 1s

3.5 Overflow with No Carry

Listing 5: Executing arithmetic operations to reach our desired number then adding

```
LDI R1 31
LDI R2 2
MUL R1 R2 # R1 is now 62
ADD R1 R2 # R1 is now 64
SB R1 0
LB R2 0 # R2 is now 64
ADD R1 R2 # should cause overflow but no carry
```

The second test for overflow and carry was overflow with no carry. This occurs when the addition of two numbers gives us a negative, but there was no carry over at the MSB. This test failed at the first run, as the carry flag was 1, but it was supposed to be 0. This was due to us using signed integers to check for carry, and after making them unsigned, we were able to detect

carry correctly for this test. The sign bit is also 0, as we the expected result of adding two positive numbers is positive. The execution of the final add instruction can be seen below.

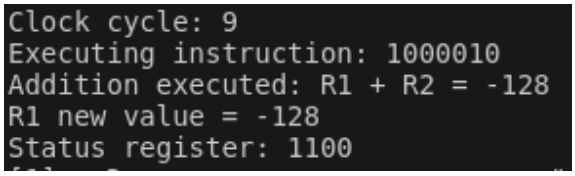


Figure 8: Overflow is 1, but carry is 0

3.6 No Overflow with Carry

Listing 6: Executing arithmetic operations to reach our desired number then adding

```
LDI R2 -32
LDI R1 2
MUL R2 R1 # R2 is now -64
SB R2 0
LB R1 0
LDI R3 -1
MUL R1 R3 # R1 is now 64
ADD R1 R2 # should cause carry but no overflow
```

The third and final test for overflow and carry was no overflow with carry. This occurs with adding two numbers and getting the correct result, but still getting a carry due to the nature of two complement's. This test gave us our expected result, which was a carry flag of 1 and an overflow flag of 0. The sign bit was also 0, since the result we expected was not negative. The execution of the final add instruction can be seen below.

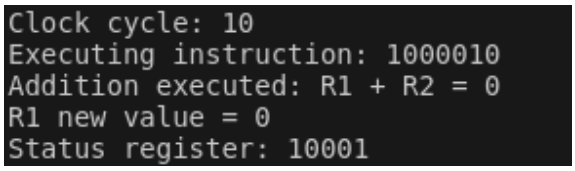


Figure 9: Carry is 1, but overflow is 0

3.7 Branch

Listing 7: Branching to instruction 6

```
LDI R0 2
LDI R1 0
BEQZ R1 2
ADD R0 R1 # should be skipped
MUL R0 R1 # should be skipped
LDI R2 10 # branch into here
MUL R0 R2 # should be 20 since both add and mul were skipped
```

Our last two tests were made for the control hazards. We first tested BEQZ by having it skip over two instructions that would affect the result of executing the final instruction. This test first failed, as we forgot to set the branch flag back to 0 once we finished disabling the decode and execute. Once this was fixed, the result came out as expected. The clock cycle is also as

expected. Since we are going to execute 3 instructions before the jump and 2 after, we expected for $3 + ((3-1) * 1)$ clock cycles $+ 3 + ((2-1) * 1)$ clock cycles, giving us a total of 9 clock cycles. Our processor executed the assembly code in the expected clock cycles. Below is an image of the final multiply instruction after branching.

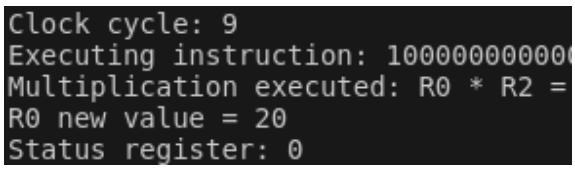


Figure 10: Result of our branch is as expected

3.8 Jump

Listing 8: Jumping to instruction 6

```
1 LDI R0 0
2 LDI R1 5
3 JR R0 R1
4 ADD R0 R1 # should be skipped
5 MUL R0 R1 # should be skipped
6 SUB R0 R1 # jump here; should be -5 since both add and mul were skipped
```

The second control hazard test was for the jump instruction. This test is the same idea as the previous test, that being the instructions skipped should affect our final instruction's result if they were executed. Running the test, we got our expected output, which was -5, as seen below. The clock cycles were also correct.

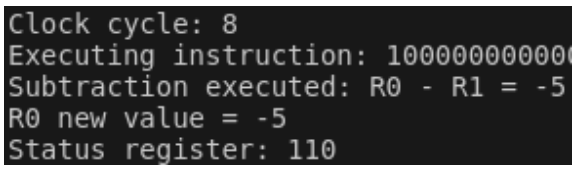


Figure 11: Result after jumping to the sub instruction