Laboratory 2 : CPUSim – addressing modes

The aim of this lab is to build upon the CPUSim register transfer level (RTL) simulation model developed in laboratory 6. For information on how to setup and use CPUSim please refer to the previous lab script. A processor's instruction-set needs to support its intended application domain i.e. implement the required functions (arithmetic, logical) and process the data types / structures used. To take this processor from a simple teaching example to one that can be used to solve real world problems we need to modify its instruction-set to support the required software structures and algorithms used. These changes will have an impact of the underlying hardware e.g. additional multiplexers to allow access to operands, therefore, when making these changes the impact on the overall system performance must always be considered. In general there is a tight coupling between an instruction-set and the underlying hardware architecture, improving processing performance of one algorithm can have a significant negative impact on another. At the end of this practical you will understand how to:

- Implement instructions that use the absolute addressing mode
- Use unconditional and conditional jump instructions
- Translate high level pseudo code such as IF, FOR and WHILE into assembly language programs.
- Use self modifying code.

Task 1

To start the CPUSim simulator left click on the Windows icon in the bottom left of the screen and type :

 `-> cpusim`

This will launch the CPUSim simulator. To load your simulation model from the previous lab click on :

```
File -> Open Machine
```

Then browse to the saved `.cpu` file. If you do not have last labs simulation model you can download a basic design (under lab script) i.e. the core processor modules, registers, micro-instructions and machine code instructions.

```
start:
     load  0x03     ;read data at address 0x03 into ACC
     add   0x01     ;increment data in ACC
     store 0x03     ;write ACC to address 0x03
```

Figure 1 : test code

A key requirement of any computer is the ability to process variables stored in memory. Therefore, the processor needs to support the absolute, or direct addressing mode i.e. `LOAD` and `STORE` instructions. To introduce these ideas consider the simple program shown in figure 1. This program loads the data stored at address 3 into the ACC, increments its values, before writing it back to memory.

Task : what two addressing modes are used in this program?

The bit-fields used to define the LOAD and STORE instructions are shown in figure 2. These instructions use the absolute addressing mode i.e. the operand they process is an 8 bit address (A bits), used to access data from memory stored at the specified "absolute address". This address is stored in the instruction register (IR) after the instruction fetch phase.

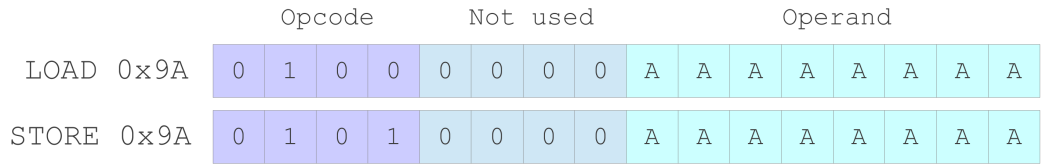


Figure 2 : LOAD and STORE instructions.

Note, LOAD and STRORE instructions are data movement instructions. These types of instructions are just as important from a data processing point of view as the number crunching instructions (+,–,×,÷). When you start to look at what happens in a computer its surprising how much time is spent moving data, therefore, efficient data transfers are the key to good processing performance.

Task : implement the LOAD and STORE instructions. If you would like to check your solutions screenshots for these instructions are shown in Appendix A.

Tips, these instructions are implemented using MemoryAccess micro-instructions, to transfer data to and from the ACC and memory.

Note, in the simulator you can only perform these memory access micro-instructions using registers. Therefore, you will need to implement another register-to-register transfer micro-instruction to load the lower 8 bits of the instruction register (IR) into the addr "register", this can then be used as the address within the micro-instruction. In the actual hardware this "register" is simply implemented using wires, selecting IR(7:0), as no memory functionality is required i.e. its a simple bit-slice operation.

Task : enter the test program shown in figure 1. Execute this code and confirm that the instructions are working correctly i.e. the data stored at address 0x03 is incremented.

```
OFFSET EQU 1
start:
     load  A        ;move data at address 0x03 into ACC
     add   OFFSET   ;increment data in ACC
     store A        ;move ACC to address 0x03

A:
     .data 1 0x0000
```

Figure 3 : test code

The test code in figure 1 can also be represented using the code in figure 3. Here, rather than using the "raw" absolute address or data values we can us the symbolic labels e.g. variable "A", or the constant OFFSET. The assembler automatically replaces the label "A" with the next free address in memory i.e. memory location

0x03. The initial value of this variable (address) can be define using the `.data` assembler directive.

Note, all labels end with a "**:**" e.g `START` and `A`. Constants such as `OFFSET` are not assigned space in data memory. For more information on these click on:

```
Help -> General CPUSim Help
```

Then browse to :

```
CPUSim Help -> Machine Specifications
            -> Assembly Language -> .data Statements
                                 -> EQUs
```

Task : modify your program to match the code shown in figure 3. Execute this code and confirm that the instructions are working correctly i.e. the data stored at address 0x03 is incremented.

Note, to ensure that this "new" code is loaded into memory, deselect `Debug Mode` from the `Execute` pull-down menu, then click on `Reset everything`. Re-assembly the program and load into memory.

The key advantage of using symbolic labels is that as your program's size changes you do not need to keep track of the absolute address at which each variable is stored. These are automatically assigned the next free memory location after the program.

Task : modify the initial data value stored in variable `A` to the value 0x1234, as shown below. Re-assembly and load this new program. What will be the final value stored in `A`? Run this program to confirm your answer. Can you see why it is not 0x1235?

```
     .data 1 0x1234
```

Task : rewrite the test code to match that shown in figure 4. What does this "program" do? Re-assembly and load this new program. Execute this code and confirm your suspicions.

```
start:
     .data 1 0x4003
     .data 1 0x1001
     .data 1 0x5003
     .data 1 0x1234
```

Figure 4 : test code

The previous task clearly shows that from the processor's point of view there is no difference between instructions and data i.e. a binary string stored in memory could represent data or an instruction, how this string is processed is dependent on whether the processor is fetching the string as an instruction, or decoding it as an operand. We will come back to this idea in task 3.

Task 2

Variables used in a program are typically stored in registers or external memory. As the SimpleCPU processor only has one data register the majority of variables need to

be stored in memory i.e. the ACC will be overwritten with the result of the last instruction. Consider the pseudo code below:

```
A = A + B
```

Task : if variables A and B are data values stored in memory can you implement this code using the current instruction-set?

Not to spoil the surprise, but the answer is no, currently the only ADD instruction we have uses immediate addressing, one operand must be hard-coded before runtime i.e. a constant, rather than a variable. To support these types of operations the following instructions can be added:

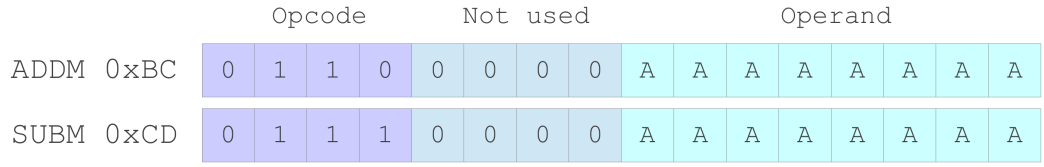


Figure 5 : ADDM and SUBM instructions.

These instructions use the absolute addressing mode i.e. an 8 bit address (A bits), used to accesses data stored in memory, that is then added / subtract to the ACC.

Note, to help machine-code "readability" the top two bits of the opcode field for an instruction using the absolute addressing mode is always "01".

Task : implement the ADDM and SUBM instructions. If you would like to check your solutions screenshots for these instructions are shown in Appendix B.

Tips, these instructions are implemented using MemoryAccess micro-instructions. The data stored at the specified absolute address is read and then added / subtracted to / from the current ACC value, the result is then stored in the ACC i.e. overwriting the old ACC value.

Note, in the simulator you can only perform arithmetic micro-instructions using registers. Therefore, you will need to implement a new MemoryAccess micro-instruction to transfer the value accessed from memory into the data "register", this can then be used as the second operand in the new arithmetic micro-instructions. In the actual hardware this "register" is simply implemented using wires, selecting the data-out bus from memory: MEM(7:0), as no memory functionality is required i.e. again its a simple bit-slice operation.

Task : rewrite the test code to match that shown in figure 6. Knowing that each instruction and variable takes one memory locations, what memory addresses are used to store the variables A, B and C? What is the final value of variable A? Re-assembly and load this new program. Check each instruction's machine code in the MEM panel to determine each variables address, can you spot these addresses within the instructions? Then execute this code to confirm the final result.

The absolute addressing mode instructions supported by the SimpleCPU processor are shown in figure 7.

```
start:
     load  A         ;zero ACC
     addm  B         ;add variable B to ACC
     subm  C         ;subtract variable C from ACC
     store A         ;write ACC to variable A

A: .data 1 0x0000
B: .data 1 0x0005
C: .data 1 0x0003
```

Figure 6 : test code

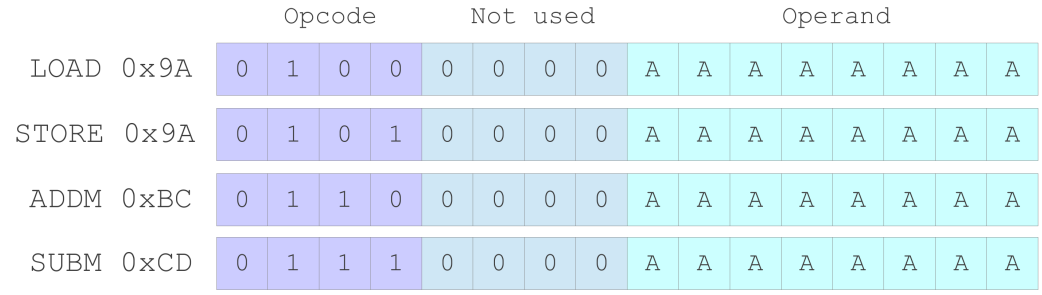


Figure 7 : absolute addressing mode instructions

Task 3

Software algorithms typically use loops to implement the desired sequence of operations, consider the pseudo code below:

```
FOR X IN RANGE 0 TO 10          WHILE A < 123
LOOP                            LOOP
     DO SOMETHING                    DO SOMETHING
END LOOP                        END LOOP
```

A `FOR` loop repeats an operations for a specified number of iterations, whilst a `WHILE` loop will repeat an operation until a specific event occurs.

To control the flow of instruction executed, the SimpleCPU processor needs to support jump instructions i.e. instruction that will change the program counter (`PC`) to a specified address in memory.

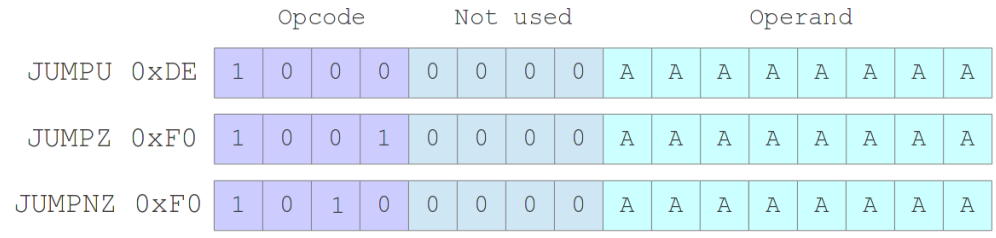


Figure 8 : `JUMP` instructions

The flow control instructions supported by the SimpleCPU processor are shown in figure 8.

Note, to help machine-code "readability" the top two bits of the opcode field for a jump instruction is always "10".

These instructions use the absolute addressing mode i.e. an 8 bit address (A bits), used to specify the target location of the next instruction. There are two types of jump instructions: conditional (JUMPZ, JUMPNZ) and unconditional (JUMPU).

Unconditional jumps always change the address stored in the program counter (PC) to the absolute address specified in the instruction i.e. the address of the next instruction to be executed. Conditional jumps are conditional on the state of the ACC i.e. a jump occurs if it is zero (JUMPZ) or not zero (JUMPNZ). If the jump condition is not true e.g. ACC=1 and JUMPZ executed, the PC is simply incremented to the next address PC=PC+1.

Task : implement the JUMPU , JUMPZ and JUMPNZ instructions. If you would like to check your solutions screenshots for these instructions are shown in Appendix C.

Tips, these instructions are implemented using transferRtoR and test micro-instructions. The JUMPU simply overwrites the PC with the absolute address stored in the IR. The conditional jumps are dependent on the value stored in the ACC, the test micro-instruction can test for these conditions and then skip (omission field) one or more micro-instructions to prevent the PC from being updated.

```
start:
     load  start    ;move data stored at START into ACC
     add   0x01     ;increment data in ACC
     store start    ;move ACC to address START
     jump  start    ;jump back to START of program
```

Figure 9 : test code

Task : rewrite the test code to match that shown in figure 9. What does this program do? Execute this code and confirm your suspicions. Step through this code such that the ADD instruction is executed three times. What does the "LOAD" instruction do after the first loop? What is the final result in the ACC?

Hint, check the machine-code stored in memory. What is stored in the ACC at the end of the LOAD instruction. What is written to memory at the end of the STORE instruction? What instructions are actually executed?

The previous task is an example of self modifying code i.e. a program that re-writes itself at runtime i.e. LOAD transformed into a MOVE. To state the obvious a program that rewrites itself is not a recommended programming technique, but it does help reduce hardware complexity and can also produce very compact code. As a result it was commonly used in first generation computers.

The assembly language implementations of the WHILE and FOR loops are shown in figures 10 and 11. Both implement multiplication through repeated addition.

Note, the final jump instruction "jump end", causes an infinite loop, to prevent the processor from "executing" the data that follows the program i.e. as there is no operating system to rollback to we need to ensure that the program "stops" when it is finished.

Task : re-implement the FOR loop using the jump if not zero (JUMPNZ) conditional jump instruction. If you would like to check your answer one possible implementation is shown in Appendix D. The code can also be downloaded from the VLE.

PSEUDO CODE

```
START:
  Total = 0
  Count = 3
WHILE Count != 0:
  Total = Total + 10
  Count = Count - 1
END LOOP
```

ASSEMBLER CODE

```
start:
     move 0x00      ;T=0
     store T
     move 0x03      ;C=3
     store C
loop:
     jumpz end      ;While C !=0
     sub 0x01       ;C=C-1
     store 0x0e
     load T         ;T=T+10
     add 0x0a
     store T
     load C
     jumpu loop     ;repeat
end:
     jumpu end

T:   .data 1 0      ;TOTAL
C:   .data 1 0      ;COUNT
```

Figure 10 : WHILE loop test code

PSEUDO CODE

```
START:
  Total = 0
FOR I IN RANGE 0 TO 3
  Total = Total+10
END LOOP
```

ASSEMBLER CODE

```
start:
     move 0x00      ;T=0
     store T
     store I        ;I=0
loop:
     load I         ;FOR LOOP
     add 0x01
     store I
     sub 0x03       ;check exit
     jumpz end
     load T         ;T=T+10
     add 0x0a
     store T
     jumpu loop     ;repeat
end:
     jumpu end

T:   .data 1 0      ;TOTAL
I:   .data 1 0      ;I
```

Figure 11 : FOR loop test code

Conditional jump instructions are also used to implement decision points in a program i.e. relational operators. Consider the pseudo shown in figure 12. This program adds two variables together A and B, then evaluates if the total is “Low” (10 or less), “Middle” (between 11 and 99), or “High” (100 or more). Also, if the total is zero the variable Z is set. To implement the greater-than and less-than relational operators we need to define new conditional instructions that use the CARRY flag.

PSEUDO CODE

```
START:
  Low = 0
  Middle = 0
  High = 0
  Zero = 0

  Total = A + B

  IF Total = 0
  THEN
    Zero = 1
  ELSIF Total > 10
  THEN
    IF Total < 100
    THEN
      Middle = 1
    ELSE
      High = 1
    END IF
  ELSE
    Low = 1
  END IF
```

ASSEMBLER CODE

```
start:
   move 0      ;init variable
   store L
   store M
   store Z
   store H
   store T
   load A      ;perform addition
   addm B
   store T
test0:        ;is ACC 0?
   and 0xFF
   jumpnz test1
   move 1
   store Z
low:
   move 1      ;yes set LOW
   store L
   jumpu exit
test1:
   sub 10      ;is ACC < 10?
   jumpc low   ;yes set LOW
   sub 90      ;is ACC > 100?
   jumpnc high ;yes set HIGH
middle:
   move 1      ;no set MIDDLE
   store M
   jumpu exit
high:
   move 1      ;set HIGH
   store H

exit:
   jumpu exit  ;end, trap code

L: .data 1 0
M: .data 1 0
H: .data 1 0
T: .data 1 0
Z: .data 1 0
```

Figure 12 : IF test code

Relational operators are typically based around the subtract instruction:

- = : to test if the variables A and B are equal we can perform the subtraction A – B, if they are equal the result will be zero i.e. will set the zero flag.
- > : to test is variable A is greater than variable B we can perform the subtraction B – A if variable A is larger a carry will be generated i.e. will set the carry flag.
- < : to test is variable A is smaller than variable B we can perform the subtraction B – A if variable A is smaller no carry will be generated i.e. the carry flag will not be set.
- Zero : to test if variable `A` is zero we can perform a bitwise `AND`, using an operand that contains all logic 1s i.e. for the simpleCPU the value 0xFF. This logical operator will only produce a zero result if the `ACC` is zero i.e. will set the zero flag. We could also set the zero flag by performing the calculation `A+0`.

By setting or clearing the zero or the carry flag we can use the appropriate conditional jump instructions to implement the various relational operators.

Task : implement the `JUMPC` and `JUMPNC` instructions. If you would like to check your solutions screenshots for these instructions are shown in Appendix E.

Tips, these instructions are implemented using the same categories of micro-instructions as the previous jump instructions. Each test condition is again based on the `ACC`, comparing it to a fixed value.

Task 4

Data within a computer is commonly structured as an array i.e. a collection of data elements, accessed by an array index e.g. array `A` indexed by variable `I`, would be written as `A[I]`. Typically stored in memory as a linear array, data elements stored in sequential memory locations e.g. `A=[1,2,3,4]`, if the base address of the first element is address 100, this data on the SimpleCPU could be organised in memory (`MEM`) as: `MEM[100]=1`, `MEM[101]=2`, `MEM[102]=3`, `MEM[103]=4`.

Task : consider the pseudo code shown in figure 13. This program accumulates the values in array `A` i.e. 1+2+3+4=10, storing the result in the variable Total. Can you implement this pseudo code in assembly language using this looping programming structure?

```
START:
     Total = 0
     A = [1,2,3,4]
     FOR I IN RANGE 0 TO 4
          Total = Total+A[I]
     END LOOP
```

Figure 13 : test code

Not to spoil the surprise, but the answer is no, there is no addressing mode to implement this array index. Presently you can only hard code the values of 1,2,3,4 as immediate values, or you could hard code the locations of these data elements as absolute addresses, neither of these solutions allows a looping programming structure.

Possible “wrong” solutions using these techniques are show in Appendix F.

Task : if the array size was increased to 100 elements what is the advantage of using a looping programming structure compared to the solutions shown in Appendix F?

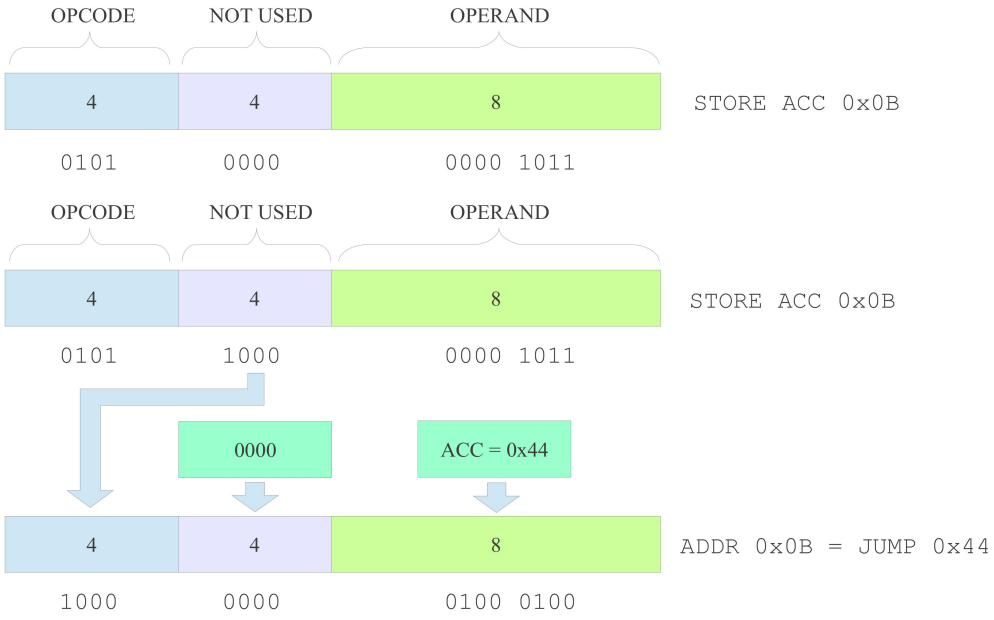


Figure 14 : new store instruction

To reduce hardware costs early computers used similar architectures to the SimpleCPU i.e. instruction-sets, processing hardware and small memory sizes. They also needed to implement these looping programming structures in order to improve code density (maximise memory efficiency). Therefore, to support these software programming structures they used self modifying code, which is always fun :).

To implement self modifying code the processor needs to write new “instructions” to memory. The instruction that writes data to memory is the `STORE` instruction, however, as we saw when we executed the test code in figure 9 as the `ACC` is only 8 bits wide, when we write data to a 16 bit memory location the high byte is set to zero. If we then use this “data” as an instruction the processor interprets this bit pattern as a `MOVE` instruction i.e. opcode “`0000`”. Therefore, we need to modify the `STORE` instruction to update the opcode bit-field to the required new instruction opcode. Consider the example in figure 14.

The four bit-field (`11` to `8`) is not used in the current `STORE` instruction, therefore, this could be used to store the new opcode e.g. “`1000`” for an unconditional jump. When the `STORE` instruction now writes to memory is will save the `ACC` to the specified absolute address, but it will also overwrite the top four bits with this new opcode value, as shown in the bottom panel of figure 14. The new store instruction is shown in figure 15.

This instruction uses the absolute addressing mode i.e. an 8 bit address (`A` bits), and a four bit immediate value i.e. an 4 bit new-opcode (`K` bits). These specify the absolute address that data will be written to: bits 0-7 are the ACC value, 8-11 are hard-coded to

"0000" and bits 12-15 are the K bits value.

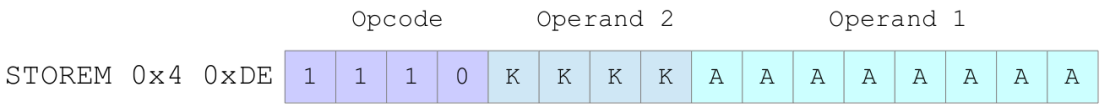


Figure 15 : new store instruction

In the SimpleCPU processor these bit-field manipulation operations are simply implement using wires as no memory functionality is required. However, in the simulator you can only perform the required micro-instructions using registers. Therefore, a new 16 bit register has to be defined.

Task : implement the STOREM instruction using the opcode value 0xE. If you would like to check your solutions screenshots for this instruction are shown in Appendix G.

Hint, define a new 16 bit register called "new". Then implement the following RTL descriptions:

```
addr        <- IR(7:0)
new(15:12)  <- IR(11:8)
new(7:0)    <- ACC
mem[addr]   <- new
end
```

One possible implementation of the loop programming structure shown in figure 13 is shown in figure 15. This implementation does differ slightly from the pseudo code. As the location of the array A is dependent on the size of the program the variable I is used as an address pointer i.e. is loaded with the base address of the array. This is then incremented to allow the program to read each element of the array.

Task : enter the program shown in figure 16. Single step through this code. Can you see how this program works? **Note**, this code can be downloaded from the VLE.

Hint, what is the difference between MOVE A and LOAD A? Do both read data from memory? Where in memory is the array A stored i.e. the values 1,2,3,4? Why is the second operand of the STOREM instruction the value 4? As the program is executed look at how the values stored in variables I, M and T change.

Task : examine the program in figure 16, how could this code be made more efficient i.e. reduce the number of instructions executed?

Hint, look at how the exit condition is performed.

Note, unless you are a virus writer self-modifying code is not a recommended programming technique. To access array data structures a modern processor uses more complex addressing modes e.g. register indirect, memory deferred, or indexed addressing modes. These will be examined in a later practical. In early computers these types of addressing modes were not possible owing to their additional hardware requirements.

```
start:
     move A          ;init index I and max count M
     store I
     store M
     move 0x00       ;zero total T
     store T

loop:
     load A          ;access array element
     addm T          ;add data to total
     store T

     load I          ;self modifying code
     add 0x01
     store I
     storem 4 loop

     load M          ;check if FOR loop is complete
     add 0x04
     subm I
     jumpz exit      ;yes, exit

     jumpu loop      ;no, repeat

exit:
     jumpu exit

A: .data 4 [1,2,3,4]      ;array
I: .data 1 0              ;array index / pointer
M: .data 1 0              ;max address check
T: .data 1 0              ;accumulator total
```

Figure 16 : looping test code

Task 5

Additional instructions can be created without modifying the hardware by combining existing instructions within a macro. Commonly used assembly language instructions, or blocks of instructions can be defined within a program using the macro's name. These are automatically replaced by the actual instructions when the program is assembled. The simplest addition to the processor's instruction set are instructions that set the `ACC` to a new value e.g. `CLR` or `SET`. The required macros are shown below:

```
MACRO clr            MACRO set
  move 0                move 0xFF
ENDM                 ENDM
```

Now, if we use the new 0-operand "instruction" `clr`, the assembler will automatic replace this text with `move 0`. Shift instructions are normally implemented using specialised hardware e.g. a barrel shifter, moving the bit pattern in the `ACC` a specified number of positions to the left or right. Each shift to the left is the same as performing

a multiplication by two, consider the value 0x0A, decimal 10, each time it is multiplied by 2 the bit pattern (1010) is shifted to the left one position:

```
0x0A = 10 = 00001010      mul 2     00010100 = 20
                          mul 2     00101000 = 40
                          mul 2     01010000 = 80
                          mul 2     10100000 = 160
```

Multiplication on a CPU is again normally implemented using dedicated hardware, however, it can also be implemented using repeated addition :

```
15x2 = 15+15,
15x3 = 15+15+15
```

Note, this technique is fine for small values, but becomes impractical for larger values e.g. 15 x 255, which will take a little while to process. The '`l`' character is a lowercase '`L`' not a number 1.

To implement a shift left instruction the required macro is shown below:

```
MACRO sl0 n
  load n
  addm n
  store n
ENDM
```

Task : write the missing macro: `mult10` to implement the pseudo code in figure 17. If you would like to check your solutions screenshots for these instructions are shown in Appendix H.

PSEUDO CODE

```
START:
  A = B×10
```

ASSEMBLER CODE

```
start:
  mult10 a b

a: .data 1 0
b: .data 1 5
```

Figure 17 : looping test code

Hint, multiplication by 10 can be done in a couple of different ways, brute force or perhaps `x10` = `x8` + `x2`. Could you use the previously define `sl0` macro?

Macros help reduce coding time by reduce the amount of code written, but they do not reduce memory size i.e. the assembler will automatically cut and paste in the same code each time a macro is called. An alternative to a macro are subroutines, these allow repeated code to be written once and used many times within a program. However, to do this the processor needs to keep a track of where it needs to return to after the subroutine has completed i.e. a subroutine's first address is fixed, the main program's return address will vary, as illustrated in figure 18. This requires specialised instructions and hardware modifications to the processor. These will be examined in a later practical i.e. `CALL` / `RETURN` instructions.

In early computers to avoid these overheads a different approach was taken i.e. self-modifying code :). The first computer to use subroutines i.e. a pre-written software libraries, was EDSAC, implementing the subroutine mechanism using a "Wheeler jump", a technique named after its creator David Wheeler. In a Wheeler Jump the returning JUMP in the subroutine is modified i.e. re-written, as the program is executed, so that the program returns back to the correct instruction.

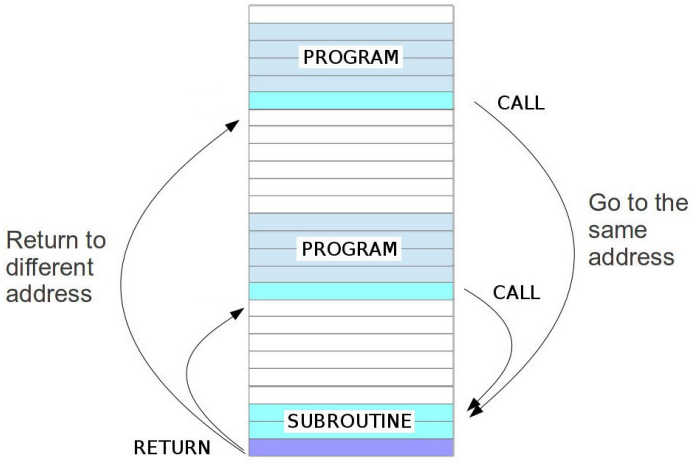


Figure 18 : subroutines

Consider the pseudo code shown in figure 19, the function `MUL( )` could be implemented as a macro, however, each time the function its used it increases the size of the instruction memory. Implementing the function as a subroutine means it is written once, used used multiple times. One possible implementation using a Wheeler Jump is shown in figure 20.

PSEUDO CODE

```
START:
  A = MUL(B,10)      ; A <- B × 10
  C = MUL(D,10)      ; C <- D × 10
```

Figure 19 : multiply function

Task : examine the assembly language program in figure 20. This program can be downloaded from the VLE. Single step this code and identify how the Wheeler Jump is implemented.

Summary

The aim of this practical was to demonstrate that instructions and data are just binary bit patterns, you can treat instructions as data, and data as instructions. This is not necessarily a good idea, but it does lead to some interesting programming techniques through the use of self-modifying code. When developing a processor's instruction-set the high level programming structures, instructions and data types used need to be identified. These are then implemented using one or more low level assembly language instructions i.e. machine-code. To help structure and reduce instruction memory size assembly language programs are typically broken down into a number of macros and subroutines. These will be examined in a later practical i.e. `CALL` / `RET` instructions using the new and improved `SimpleCPU_v1d` processor.

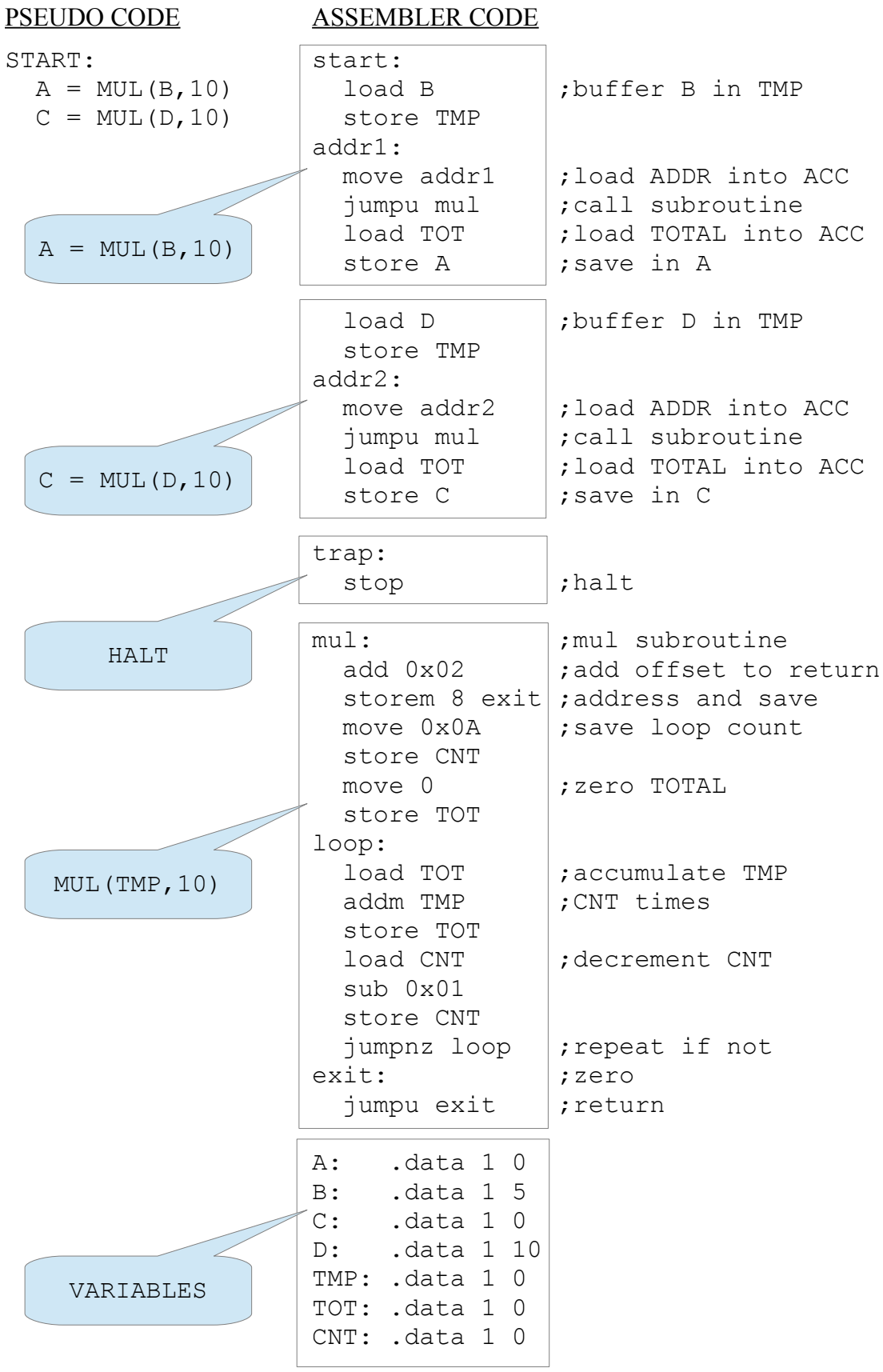
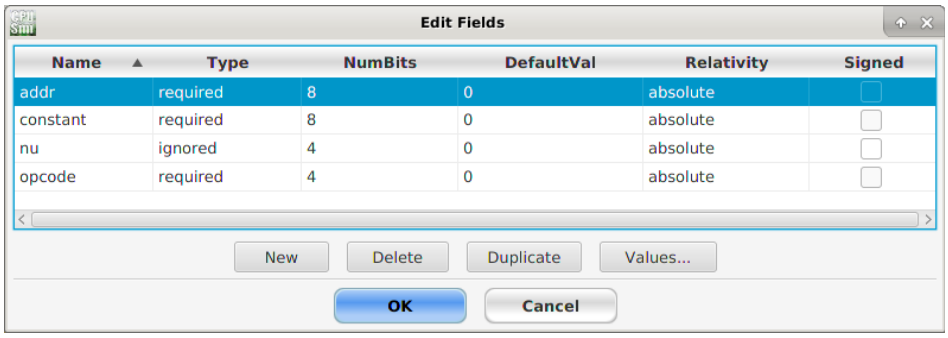
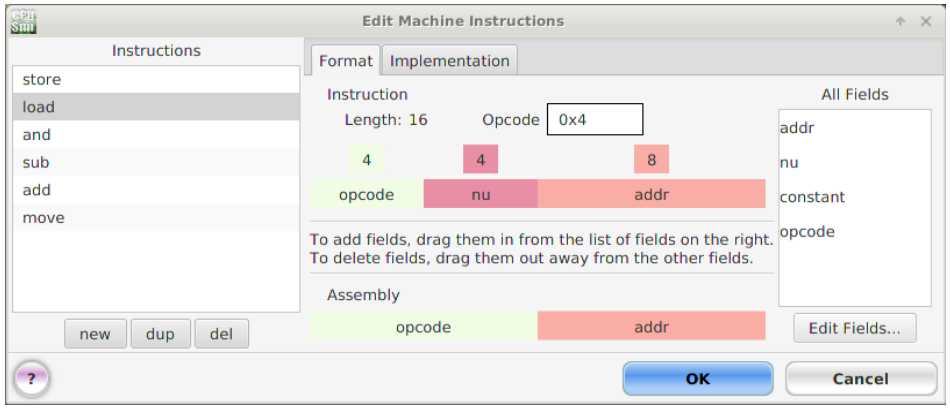
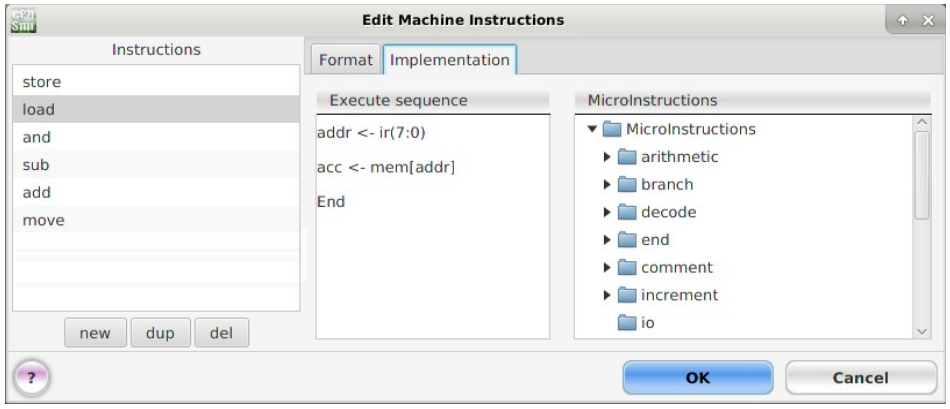


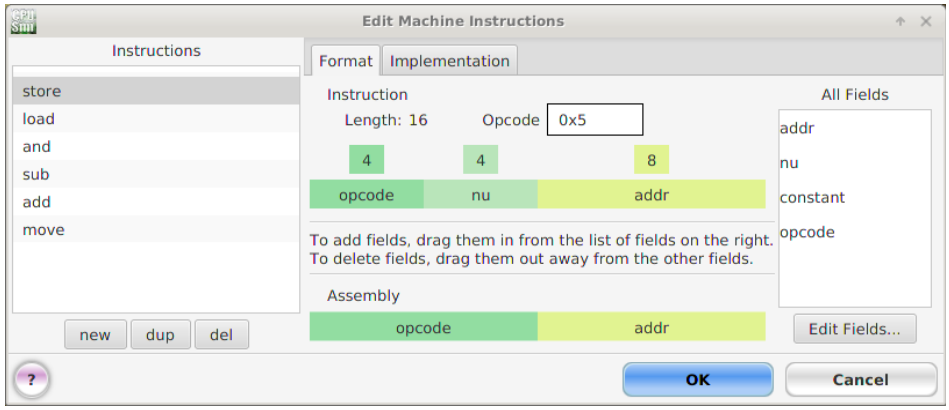
Figure 20 : multiplication subroutine

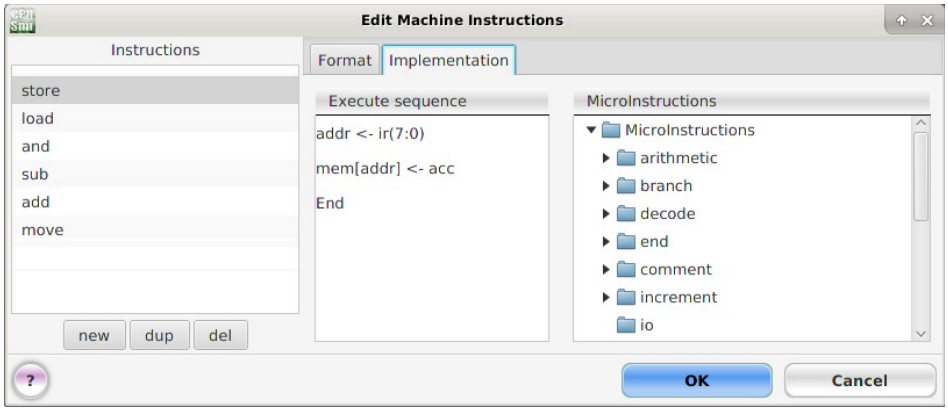
Appendix A : LOAD and STORE instruction



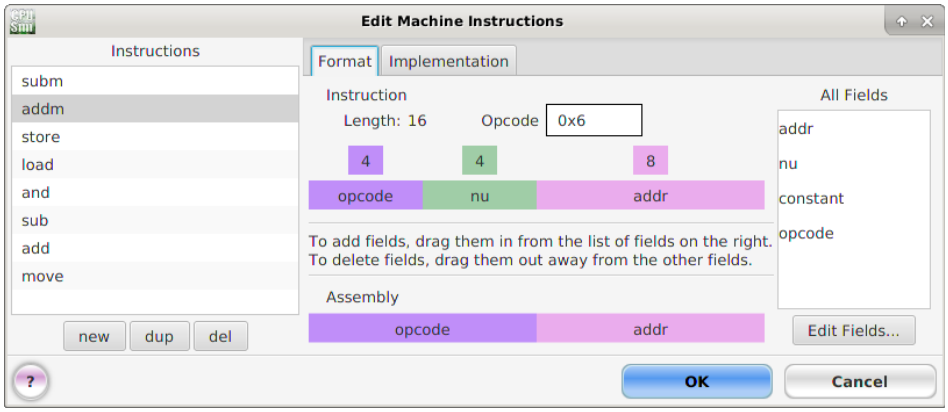


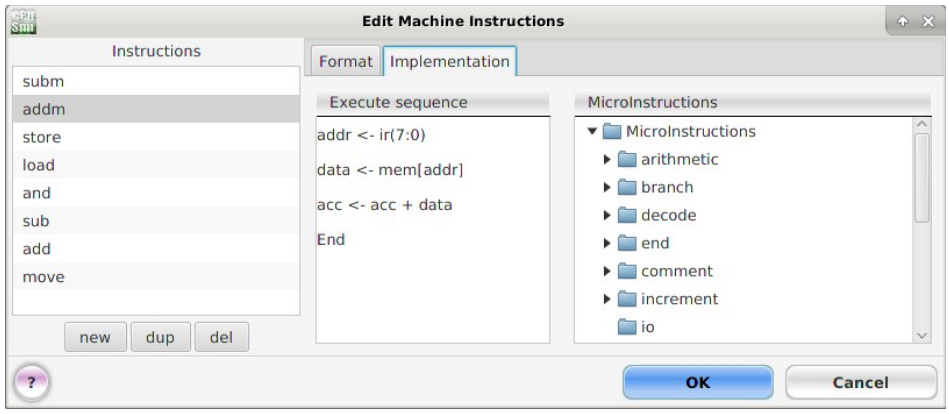


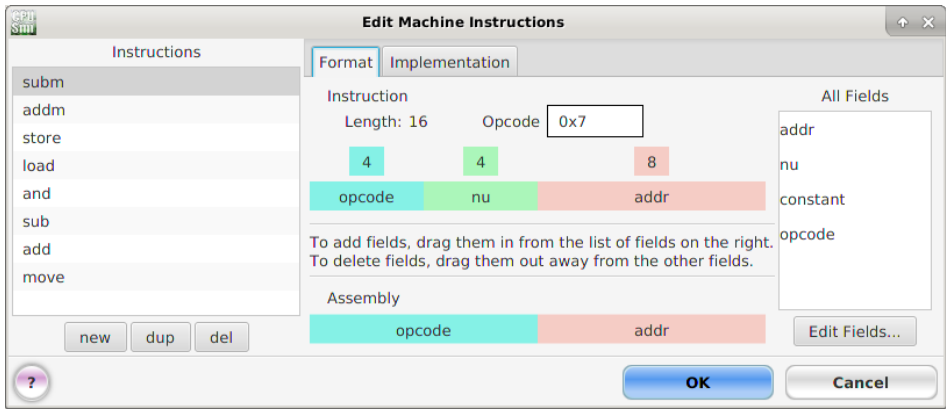


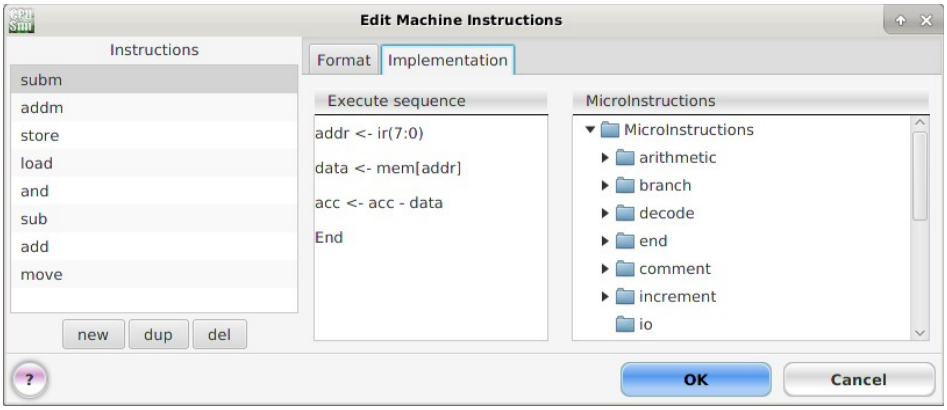


Appendix B : ADDM and SUBM instruction

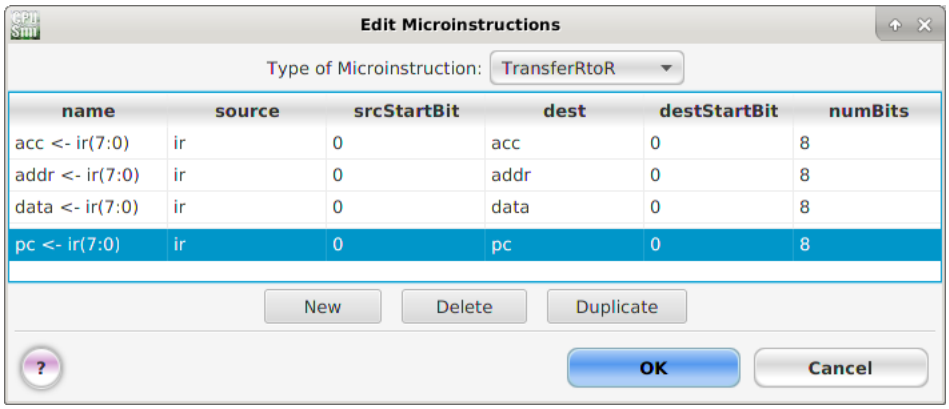


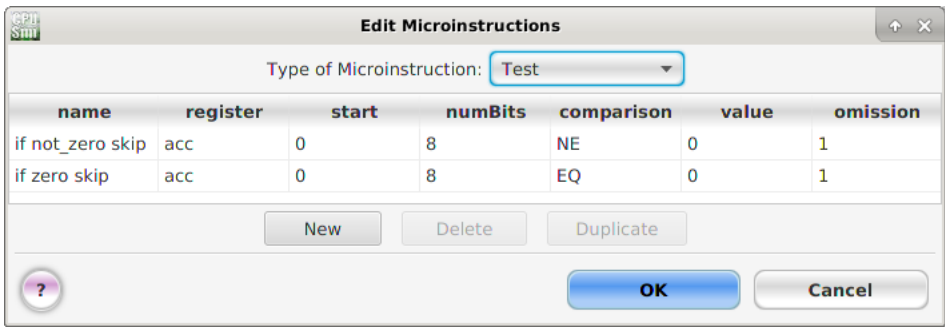


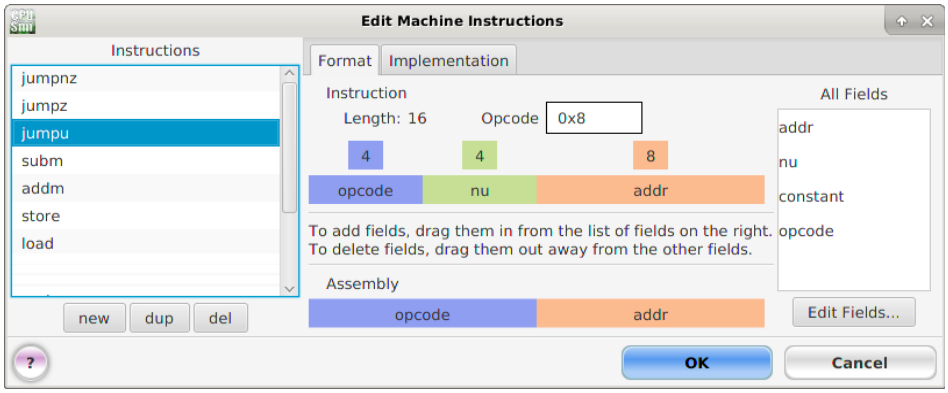


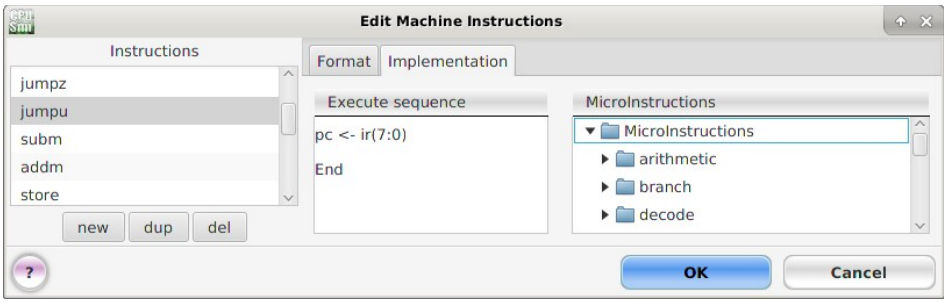


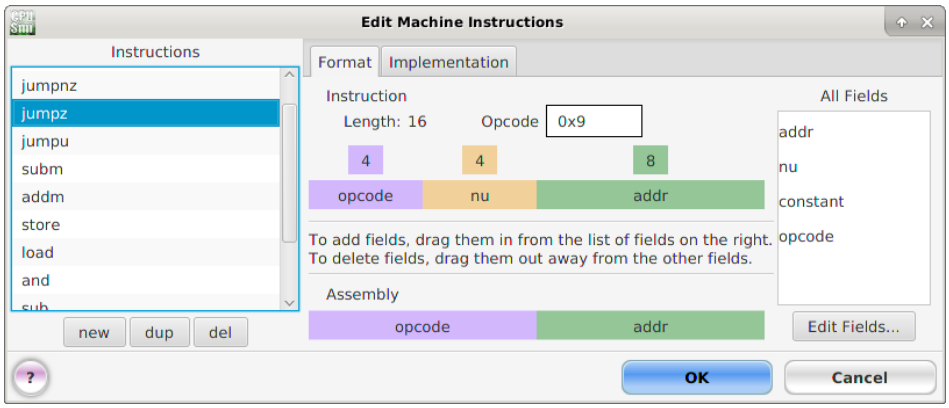
Appendix C : JUMPU, JUMPZ & JUMPNZ instructions

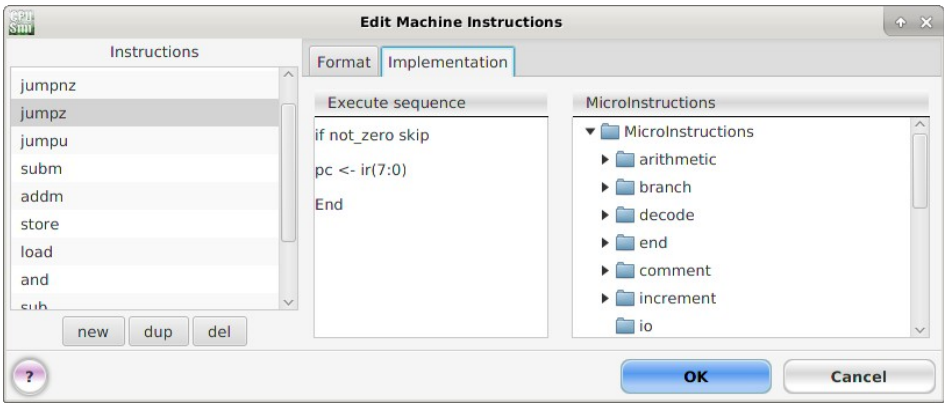


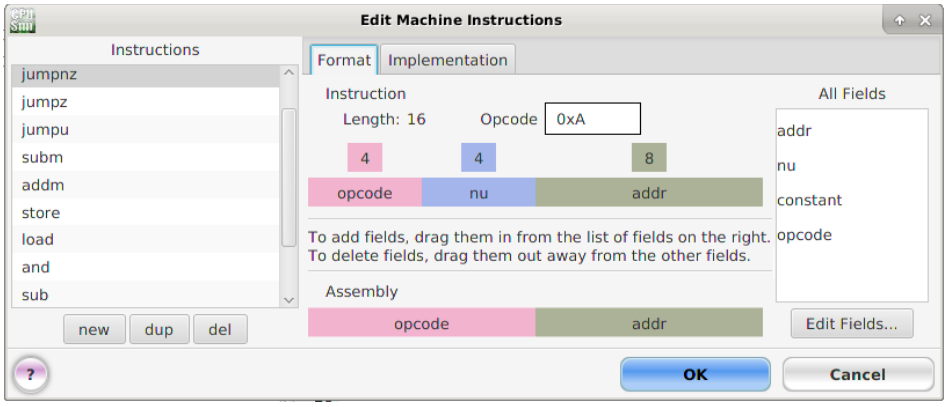


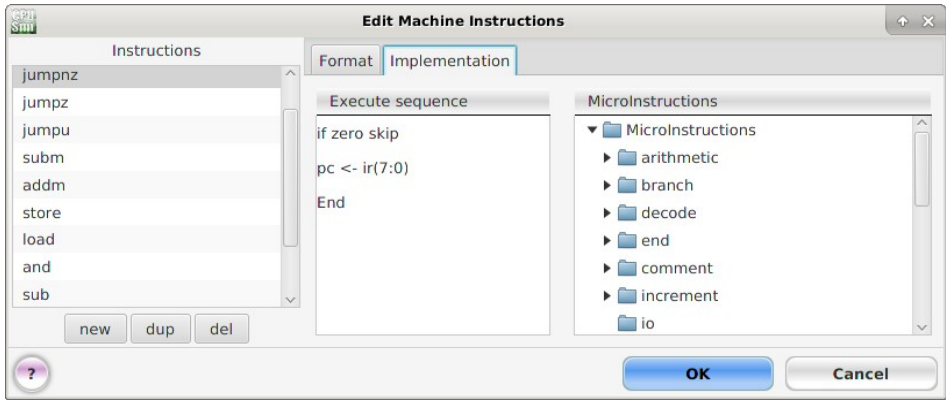



Edit Machine Instructions
Instructions
jumpz
jumpu
subm
addm
store
new
dup
del
Format
Implementation
Execute sequence
pc <- ir(7:0)
End
MicroInstructions
MicroInstructions
arithmetic
branch
decode
OK
Cancel


Edit Machine Instructions
Instructions
jumpnz
jumpz
jumpu
subm
addm
store
load
and
new
dup
del
Format
Implementation
Instruction
Length: 16
Opcode
0x9
4
4
8
opcode
nu
addr
To add fields, drag them in from the list of fields on the right.
To delete fields, drag them out away from the other fields.
Assembly
opcode
addr
All Fields
addr
nu
constant
opcode
Edit Fields...
OK
Cancel


Edit Machine Instructions
Instructions
jumpnz
jumpz
jumpu
subm
addm
store
load
and
new
dup
del
Format
Implementation
Execute sequence
if not_zero skip
pc <- ir(7:0)
End
MicroInstructions
MicroInstructions
arithmetic
branch
decode
end
comment
increment
io
OK
Cancel


Edit Machine Instructions
Instructions
jumpnz
jumpz
jumpu
subm
addm
store
load
and
sub
new
dup
del
Format
Implementation
Instruction
Length: 16
Opcode
0xA
4
4
8
opcode
nu
addr
To add fields, drag them in from the list of fields on the right.
To delete fields, drag them out away from the other fields.
Assembly
opcode
addr
All Fields
addr
nu
constant
opcode
Edit Fields...
OK
Cancel



Appendix D : FOR loop

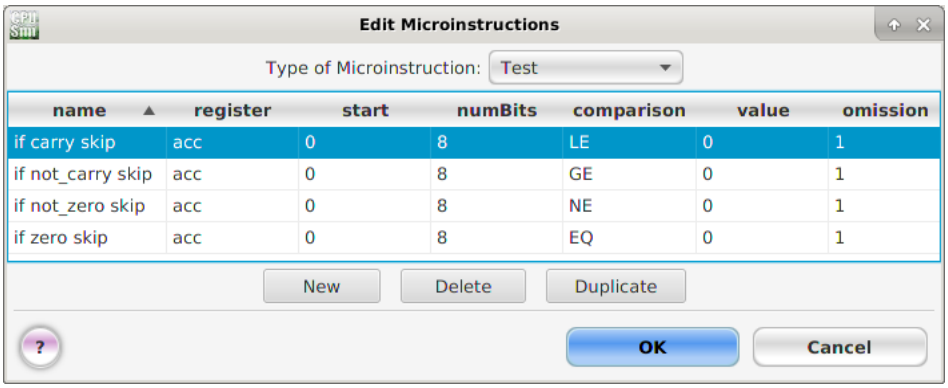
PSEUDO CODE

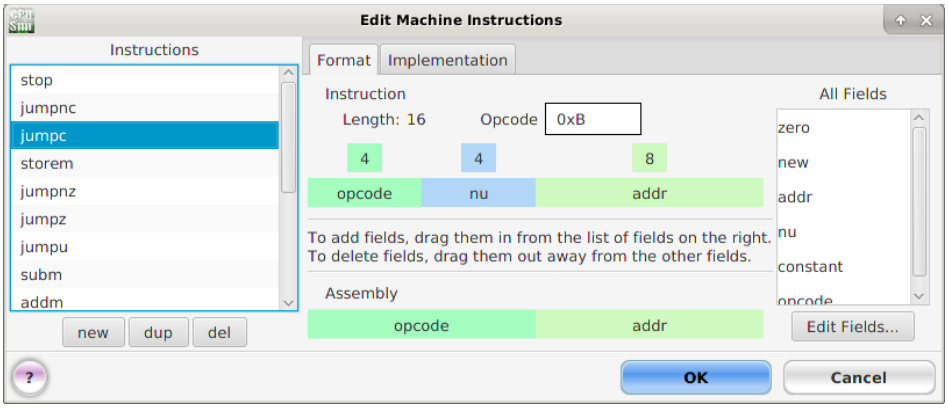
```
START:
  Total = 0
FOR I IN RANGE 0 TO 3
  Total = Total+10
END LOOP
```

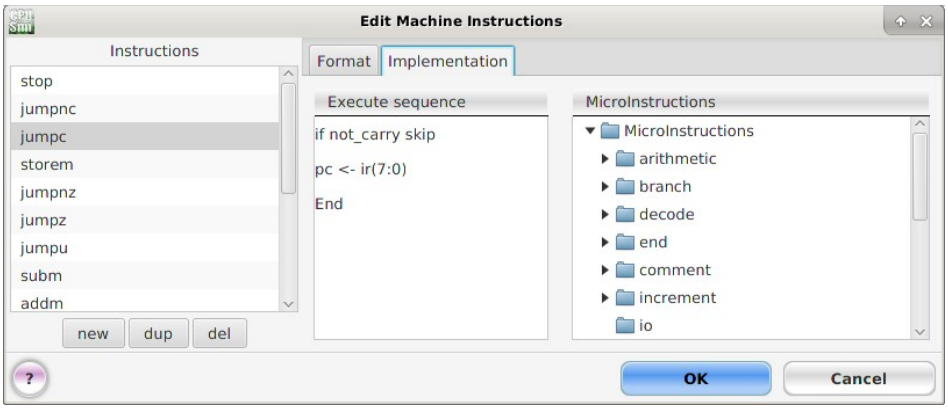
ASSEMBLER CODE

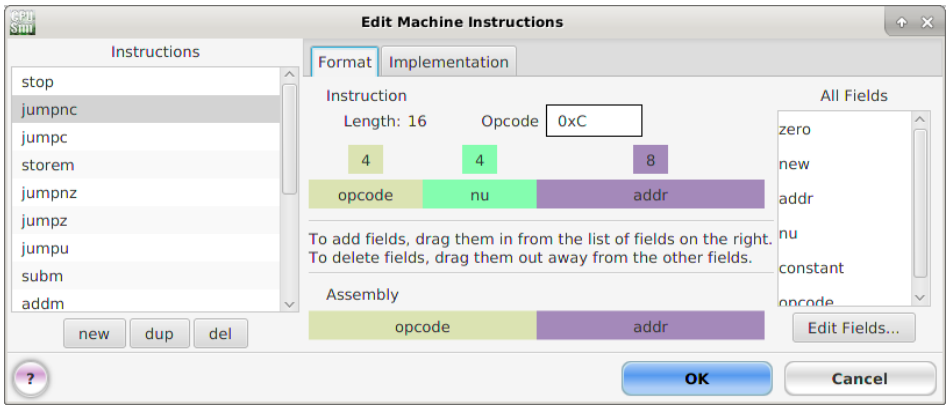
```
start:
      move 0x00       ;T=0
      store T
      store I         ;I=0
loop:
      load T          ;T=T+10
      add 0x0a
      store T
      load I          ;FOR LOOP
      add 0x01
      store I
      sub 0x03
      jumpnz loop     ;repeat
end:
      jumpu end

T:    .data 1 0       ;TOTAL
I:    .data 1 0       ;I
```

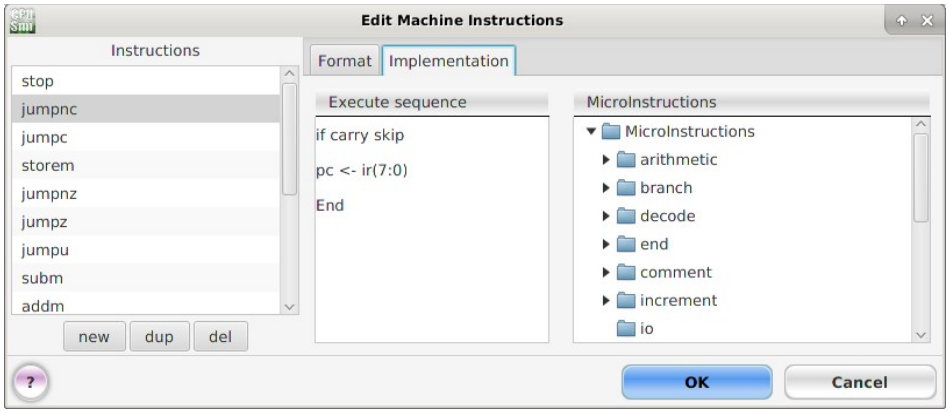
Appendix E : JUMPC and JUMPNC




Edit Machine Instructions
Instructions
stop
jumpnc
jumpc
storem
jumpnz
jumpz
jumpu
subm
addm
new
dup
del
Format
Implementation
Instruction
Length: 16
Opcode
0xB
4
4
8
opcode
nu
addr
To add fields, drag them in from the list of fields on the right.
To delete fields, drag them out away from the other fields.
Assembly
opcode
addr
All Fields
zero
new
addr
nu
constant
opcode
Edit Fields...
OK
Cancel


Edit Machine Instructions
Instructions
stop
jumpnc
jumpc
storem
jumpnz
jumpz
jumpu
subm
addm
new
dup
del
Format
Implementation
Execute sequence
if not_carry skip
pc <- ir(7:0)
End
MicroInstructions
MicroInstructions
arithmetic
branch
decode
end
comment
increment
io
OK
Cancel


Edit Machine Instructions
Instructions
stop
jumpnc
jumpc
storem
jumpnz
jumpz
jumpu
subm
addm
new
dup
del
Format
Implementation
Instruction
Length: 16
Opcode
0xC
4
4
8
opcode
nu
addr
To add fields, drag them in from the list of fields on the right.
To delete fields, drag them out away from the other fields.
Assembly
opcode
addr
All Fields
zero
new
addr
nu
constant
opcode
Edit Fields...
OK
Cancel

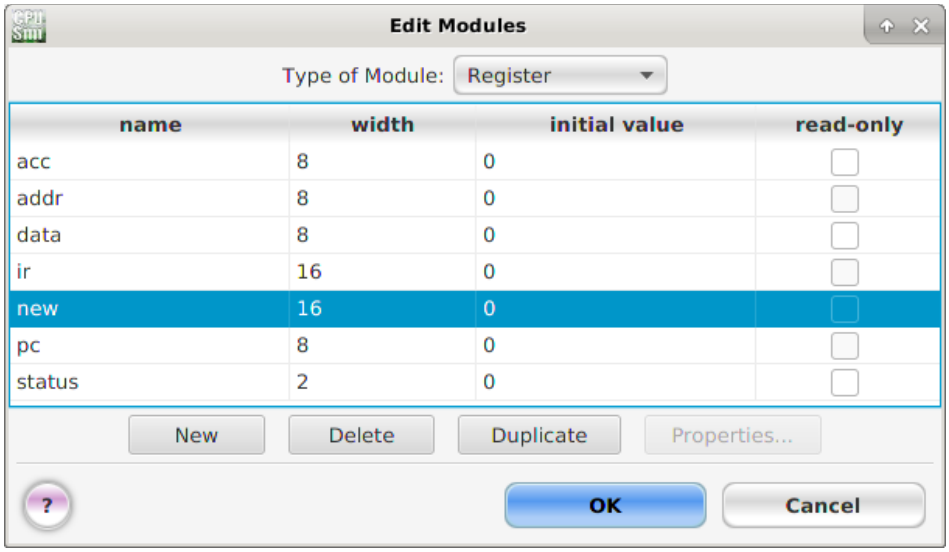

Edit Machine Instructions
Instructions
stop
jumpnc
jumpc
storem
jumpnz
jumpz
jumpu
subm
addm
new
dup
del
Format
Implementation
Execute sequence
if carry skip
pc <- ir(7:0)
End
MicroInstructions
MicroInstructions
arithmetic
branch
decode
end
comment
increment
io
OK
Cancel

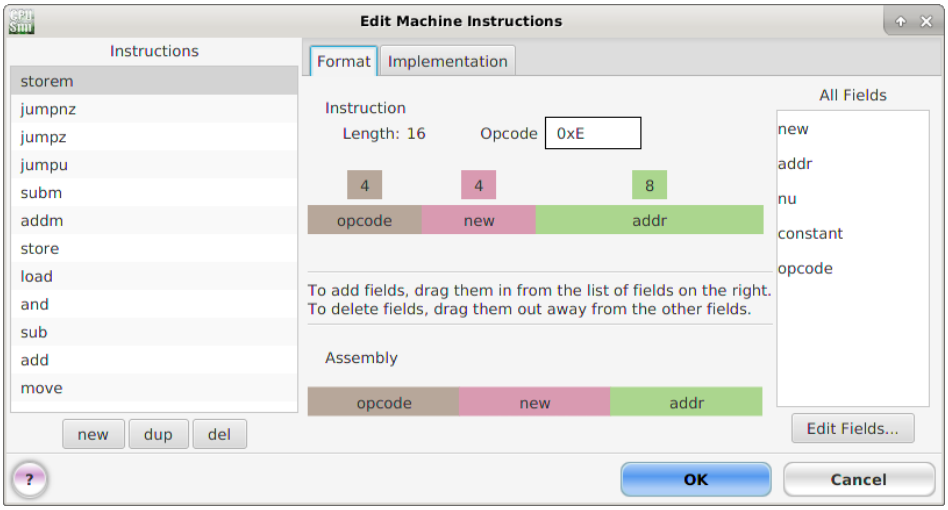
Appendix F : Simple accumulate code

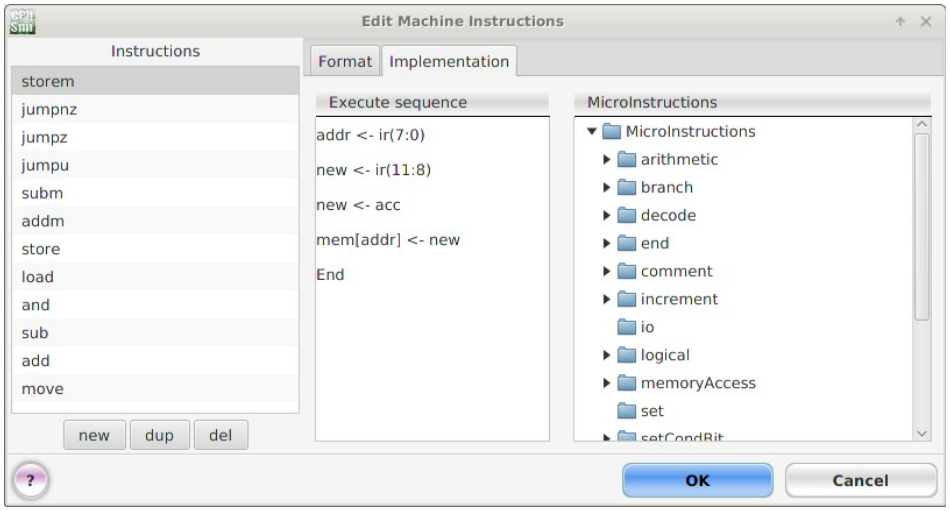
```
start:
      move  0x00
      add   0x01
      add   0x02
      add   0x03
      add   0x04
```

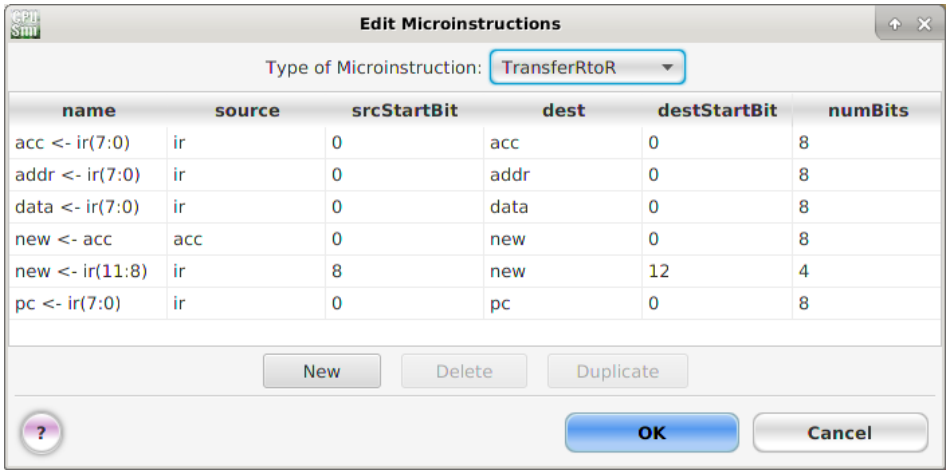
```
start:
      load A0
      addm A1
      addm A2
      addm A3

A0: .data 1 1
A1: .data 1 2
A2: .data 1 3
A3: .data 1 4
```

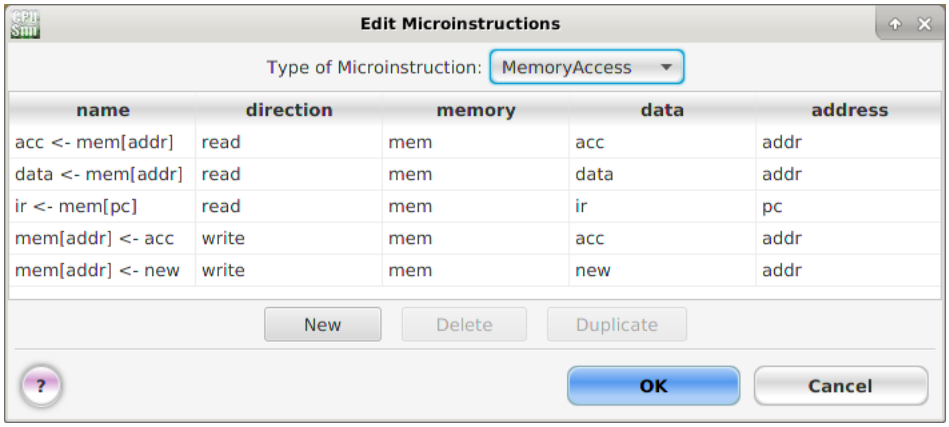
Appendix G : New STORE instruction






Edit Machine Instructions
Instructions
storem
jumpnz
jumpz
jumpu
subm
addm
store
load
and
sub
add
move
new
dup
del
Format
Implementation
Execute sequence
addr <- ir(7:0)
new <- ir(11:8)
new <- acc
mem[addr] <- new
End
MicroInstructions
MicroInstructions
arithmetic
branch
decode
end
comment
increment
io
logical
memoryAccess
set
OK
Cancel


Edit Microinstructions
Type of Microinstruction: TransferRtoR
name source srcStartBit dest destStartBit numBits
acc <- ir(7:0) ir 0 acc 0 8
addr <- ir(7:0) ir 0 addr 0 8
data <- ir(7:0) ir 0 data 0 8
new <- acc acc 0 new 0 8
new <- ir(11:8) ir 8 new 12 4
pc <- ir(7:0) ir 0 pc 0 8
New
Delete
Duplicate
OK
Cancel


Edit Microinstructions
Type of Microinstruction: MemoryAccess
name direction memory data address
acc <- mem[addr] read mem acc addr
data <- mem[addr] read mem data addr
ir <- mem[pc] read mem ir pc
mem[addr] <- acc write mem acc addr
mem[addr] <- new write mem new addr
New
Delete
Duplicate
OK
Cancel

Appendix H : MULT10

```
MACRO sl0 x
  load x
  addm x
  store x
ENDM

MACRO mult10 x y
  sl0 x
  store y
  sl0 x
  sl0 x
  addm y
  store y
ENDM

Start:
  mult10 a b

a: .data 1 2
b: .data 1 0
```