Report of HCL Mini Project

Computer Architecture

Academic Year: 2024–2025

Group A1

Conducted by

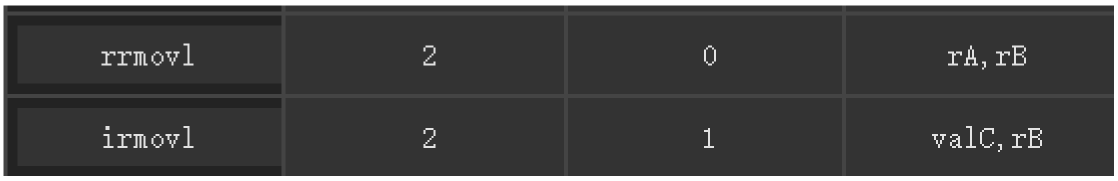
XING Jiayu
TRUONG Thien An

REPORT MINI PROJECT

Exercise1

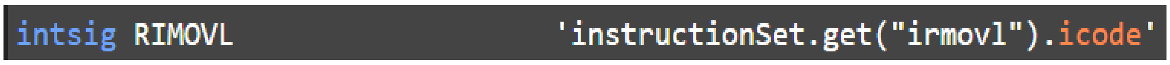
We read the instruction of ex1, the first thing we did is to change the icode of irmovl to the same with rrmovl and we modified the ifun too.

Question1 requires removing irmovl in every line in HCL and adding “|| (icode == RRMOVL && ifun == 1)” as a replacement of irmovl and the end of every line that has been modified.



rrmovl	2	0	rA, rB
irmovl	2	1	valC, rB

Question2 requires modifying the simulator's HCL code so the new versions of `rrmovl` and `irmovl` behave the same as the originals. We replaced the notation of these two constants with one single constant RIMOVL. Then we replaced RRMOVL with RIMOVL.



```
intsig RIMOVL                  'instructionSet.get("irmovl").icode'
```

but in the editor part, we ought to keep using the syntax irmovl and rrmovl despite the redefinition in HCL code.

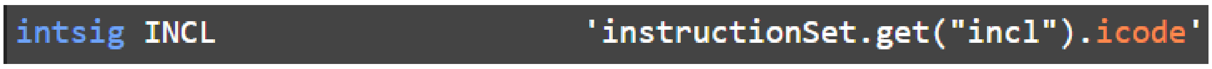
Exercise2

We added two new instructions :incl and decl, to increment or decrement the value of a register by 1.These instructions are equivalent to iaddl 1, %reg and isubl 1, %reg, respectively, but are more compact.

Question123

Step1 : Definition of many notations

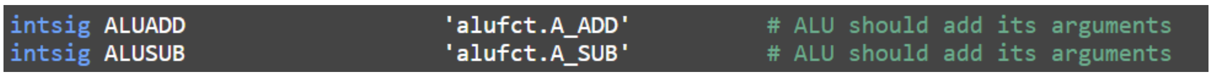
We defined a new opcode called INCL for both instructions, We used ifun = 0 for incl, and ifun = 1 for decl.



```
intsig INCL                    'instructionSet.get("incl").icode'
```

We defined in the HCL this line in saying that we only need the icode of the instruction incl,

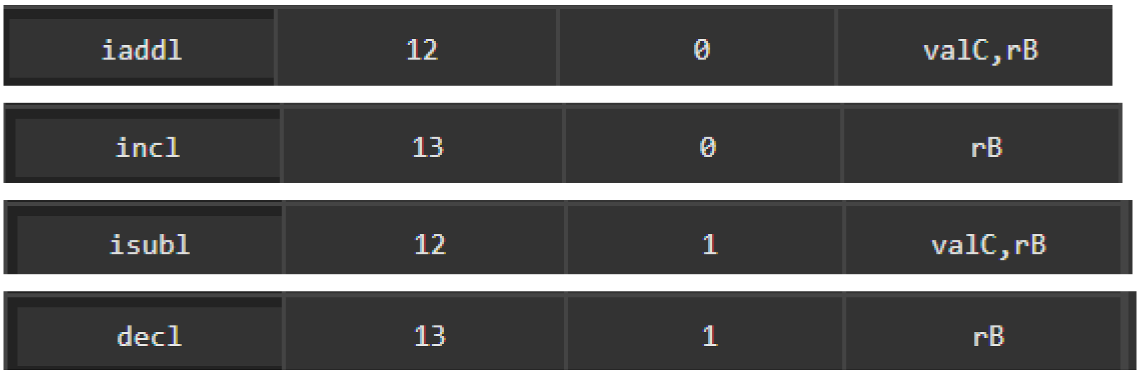
We defined ALUSUB in the vision of being able to use it later



```
intsig ALUADD                  'alufct.A_ADD'         # ALU should add its arguments
intsig ALUSUB                  'alufct.A_SUB'         # ALU should add its arguments
```

Initially, we tended to use rA as the main parameter for the incl and decl instruction. But gradually, we recognised that to simplify the code with that same syntax from “iaddl”, we decided to put rB to the main parameter in the instruction set, for these following conveniences:

But before that, let take a look at these:



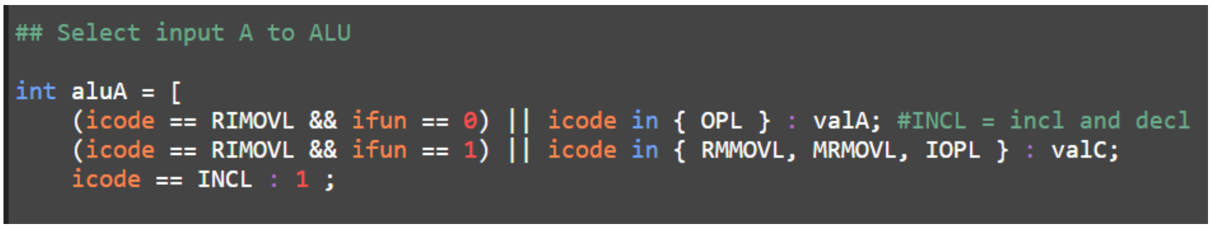
iaddl	12	0	valC,rB
incl	13	0	rB
isubl	12	1	valC,rB
decl	13	1	rB

Let’s study the logic behind the scene:

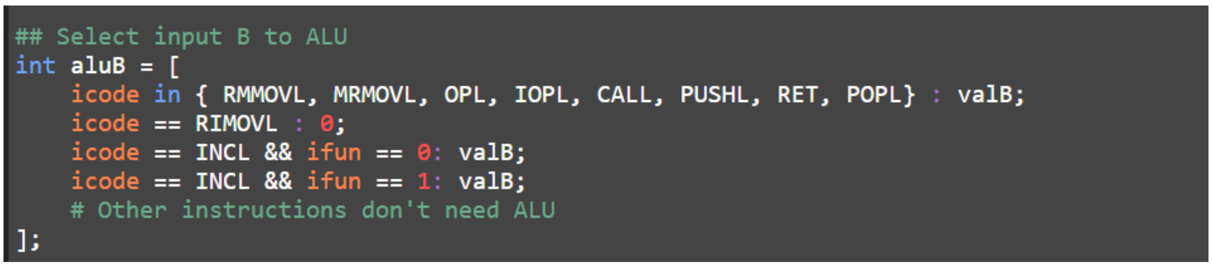
Command	icode	ifun	arg1 (aluA)	arg2 (aluB)
iaddl	12	0	valC	rB
incl	13	0	1	rB (*)
isubl	12	1	valC	rB
decl	13	1	1	rB (*)

*the choice of rB will be justified in the following part

In iaddl and isubl there is valC that will be inputted by user while incl and decl is implicitly declared as “1”. Also valC and 1, and 4 commands using **rB** will be respectively transferred to ALU as aluA and **aluB**. **Overall : valC and 1 ->aluA, rB->aluB**

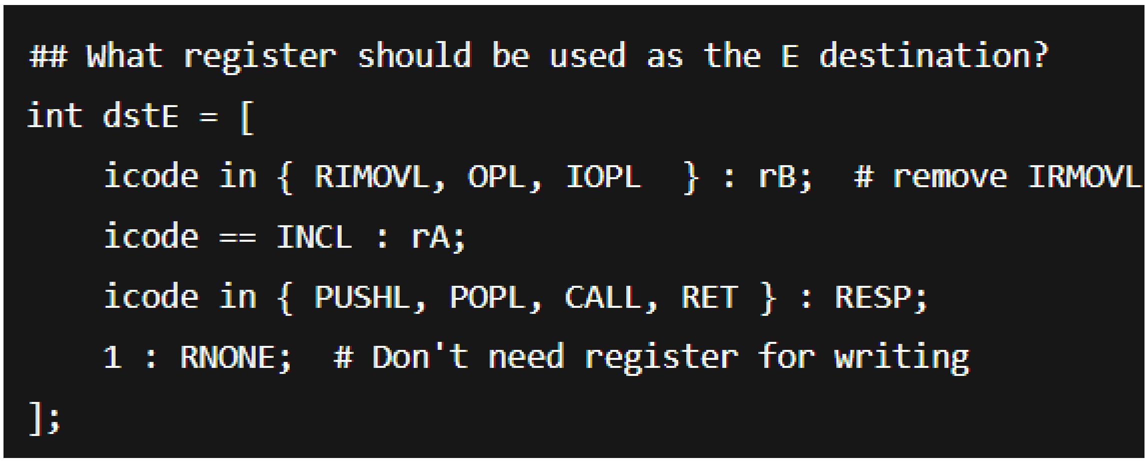


```
## Select input A to ALU

int aluA = [
    (icode == RIMOVL && ifun == 0) || icode in { OPL } : valA; #INCL = incl and decl
    (icode == RIMOVL && ifun == 1) || icode in { RMMOVL, MRMOVL, IOPL } : valC;
    icode == INCL : 1 ;
```



```
## Select input B to ALU
int aluB = [
    icode in { RMMOVL, MRMOVL, OPL, IOPL, CALL, PUSHL, RET, POPL} : valB;
    icode == RIMOVL : 0;
    icode == INCL && ifun == 0: valB;
    icode == INCL && ifun == 1: valB;
    # Other instructions don't need ALU
];
```

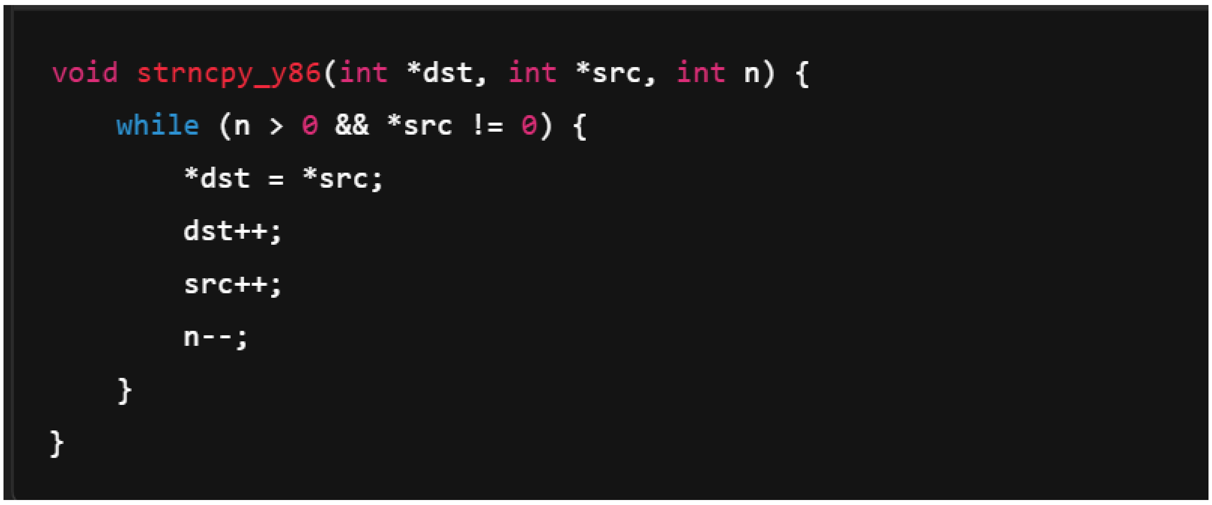
DISCOVERY: Before successfully implementing the instruction using `rB`, we discovered that using `rA` also led to correct execution. This was possible due to how certain parts of the HCL code handled operand selection. For example, in our initial implementation, we had:



```
## What register should be used as the E destination?
int dstE = [
    icode in { RIMOVL, OPL, IOPL  } : rB;  # remove IRMOVL
    icode == INCL : rA;
    icode in { PUSHL, POPL, CALL, RET } : RESP;
    1 : RNONE;  # Don't need register for writing
];
```

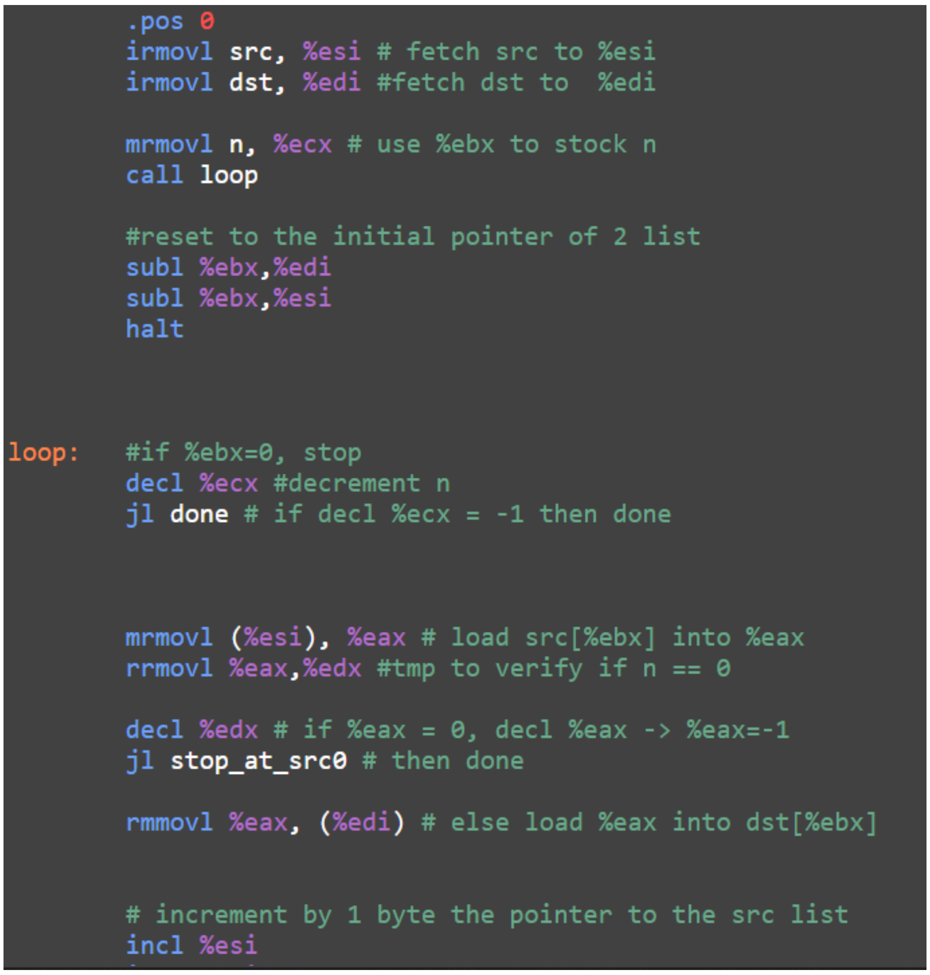
However, we ultimately decided to use rb instead of ra for the following reasons:
Uniformity and consistency: Most instructions that write results, such as irmovl, op, and iop, use rB as the destination register. Choosing rB helps maintain a consistent coding style.
Avoiding confusion: If different instructions use different register fields as destinations, it may cause misunderstandings during debugging and code review.
Improved extensibility: If we add new instructions or features in the future, following a uniform register usage pattern will reduce the cost of modification.

Question4
Here is the equivalent example of function **strncpy** (in C) which copies n elements from *list* **src** to *list* **dest,** and also will stop copying when encounter 0

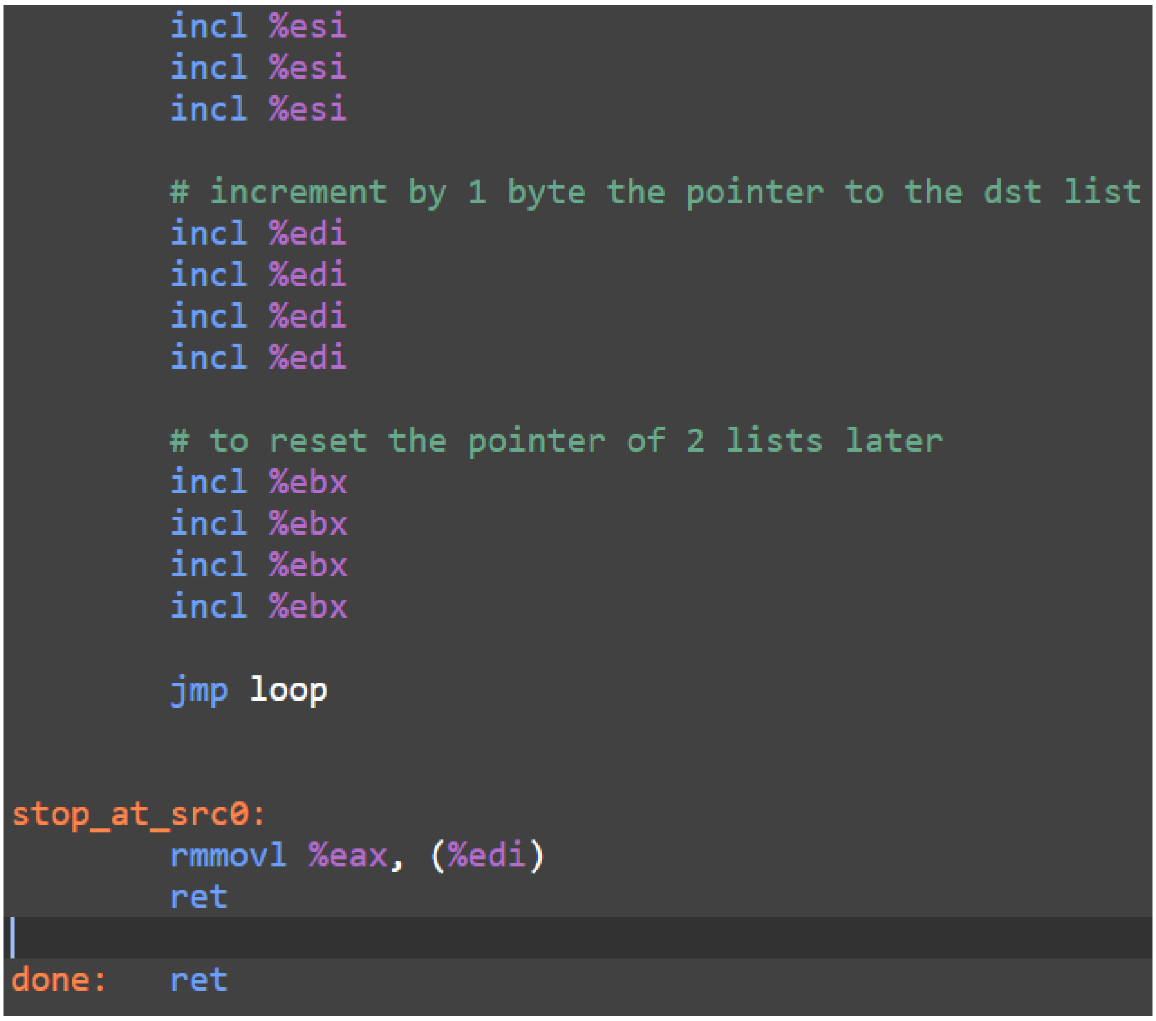


```c
void strncpy_y86(int *dst, int *src, int n) {
    while (n > 0 && *src != 0) {
        *dst = *src;
        dst++;
        src++;
        n--;
    }
}
```

Here is the adaptation to y86:

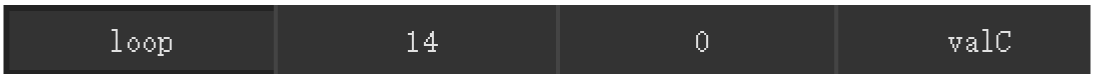


```
        .pos 0
        irmovl src, %esi # fetch src to %esi
        irmovl dst, %edi #fetch dst to  %edi

        mrmovl n, %ecx # use %ebx to stock n
        call loop

        #reset to the initial pointer of 2 list
        subl %ebx,%edi
        subl %ebx,%esi
        halt


loop:   #if %ebx=0, stop
        decl %ecx #decrement n
        jl done # if decl %ecx = -1 then done


        mrmovl (%esi), %eax # load src[%ebx] into %eax
        rrmovl %eax,%edx #tmp to verify if n == 0

        decl %edx # if %eax = 0, decl %eax -> %eax=-1
        jl stop_at_src0 # then done

        rmmovl %eax, (%edi) # else load %eax into dst[%ebx]


        # increment by 1 byte the pointer to the src list
        incl %esi
```



```
        incl %esi
        incl %esi
        incl %esi

        # increment by 1 byte the pointer to the dst list
        incl %edi
        incl %edi
        incl %edi
        incl %edi

        # to reset the pointer of 2 lists later
        incl %ebx
        incl %ebx
        incl %ebx
        incl %ebx

        jmp loop


stop_at_src0:
        rmmovl %eax, (%edi)
        ret

done:   ret
```

Exercise3
In this exercise, we focused on implementing the LOOP instruction, which is typically used to facilitate the writing of for loops where a counter is decremented. The LOOP instruction is equivalent to performing isubl 1, %ecx followed by jne label, without modifying the condition codes (ZF, SF, OF).
Question 1
We first modified the Y86 instruction set to add the new LOOP instruction.

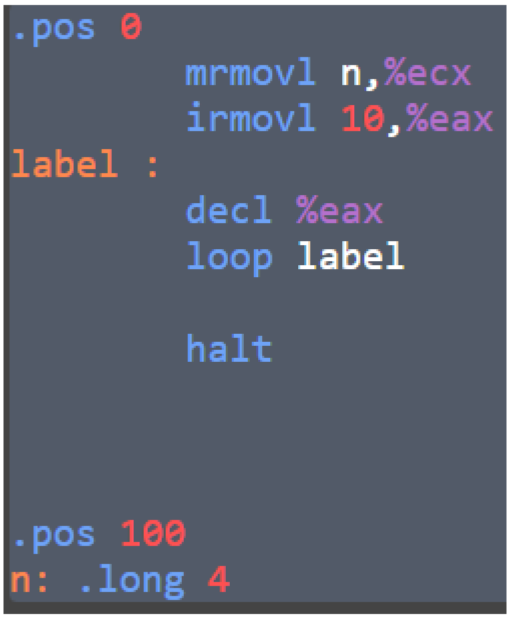


loop	14	0	valC

We assigned a new icode value dedicated to LOOP, which we chose as 0xB. We also chose an arbitrary ifun value of 0, as allowed by the exercise.

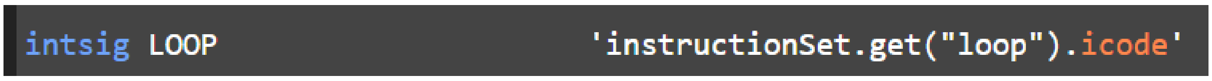
We updated the assembler to recognize the keyword loop

After modifying the assembler, we wrote the following Y86 program to test the new instruction:



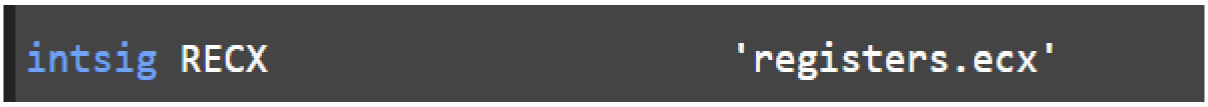
```
.pos 0
        mrmovl n,%ecx
        irmovl 10,%eax
label :
        decl %eax
        loop label

        halt



.pos 100
n: .long 4
```

We successfully assembled this program. The assembler recognized the loop instruction and generated the correct machine code.
Thus, we confirmed that the new instruction was properly handled by the assembler.
Question 2
We then modified the HCL code to implement the loop instruction execution inside the processor.
First, we added two signals:
LOOP, which indicates if the current instruction is a loop.



```
intsig LOOP                    'instructionSet.get("loop").icode'
```

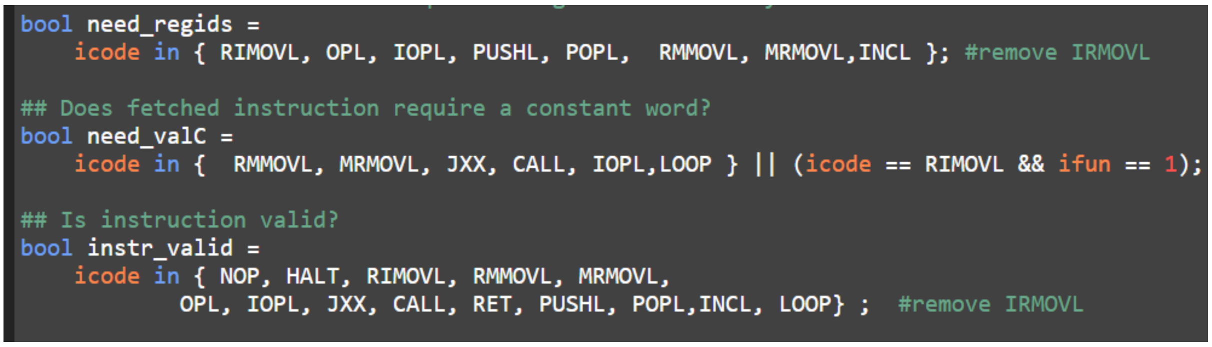
RECX, which is used to decrement the %ecx register.



```
intsig RECX                    'registers.ecx'
```

We calculate valE = valA - 1 to get the new value for %ecx, and this operation doesn’t affect the condition codes like ZF, SF, and OF. To do this, we need these following steps:

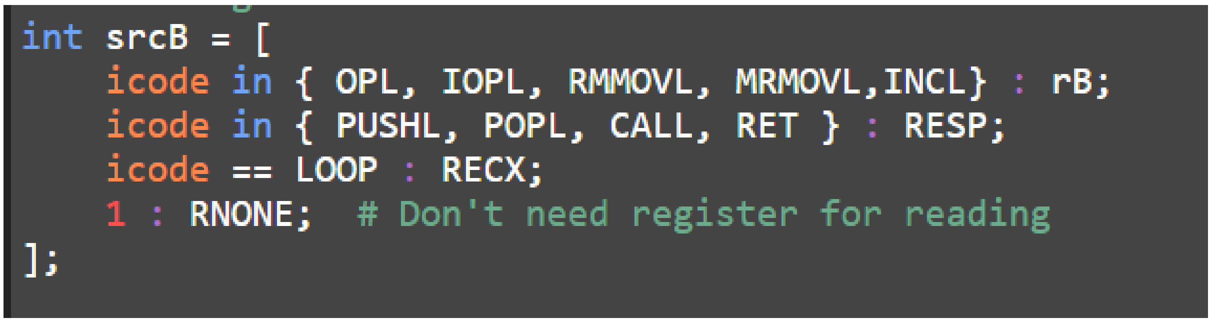
Step 1:During fetch stage, modify the HCL such that the program recognise the new instruction:



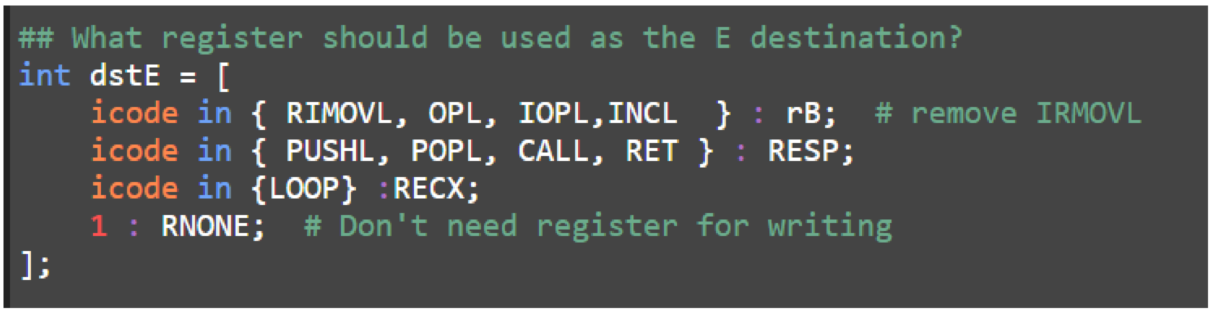
```
bool need_regids =
    icode in { RIMOVL, OPL, IOPL, PUSHL, POPL,  RMMOVL, MRMOVL,INCL }; #remove IRMOVL

## Does fetched instruction require a constant word?
bool need_valC =
    icode in {  RMMOVL, MRMOVL, JXX, CALL, IOPL,LOOP } || (icode == RIMOVL && ifun == 1);

## Is instruction valid?
bool instr_valid =
    icode in { NOP, HALT, RIMOVL, RMMOVL, MRMOVL,
            OPL, IOPL, JXX, CALL, RET, PUSHL, POPL,INCL, LOOP} ;  #remove IRMOVL
```

a. in need_regids we dont need to include LOOP since we have a specific register for it, which is %ecx
b. Still, it ll need valC as argument in this case, take a label as argument to execute the branching behavior
c. Indeed, add LOOP into instr_valid to verify it

Step2: in Decode stage, we modify srcB as below



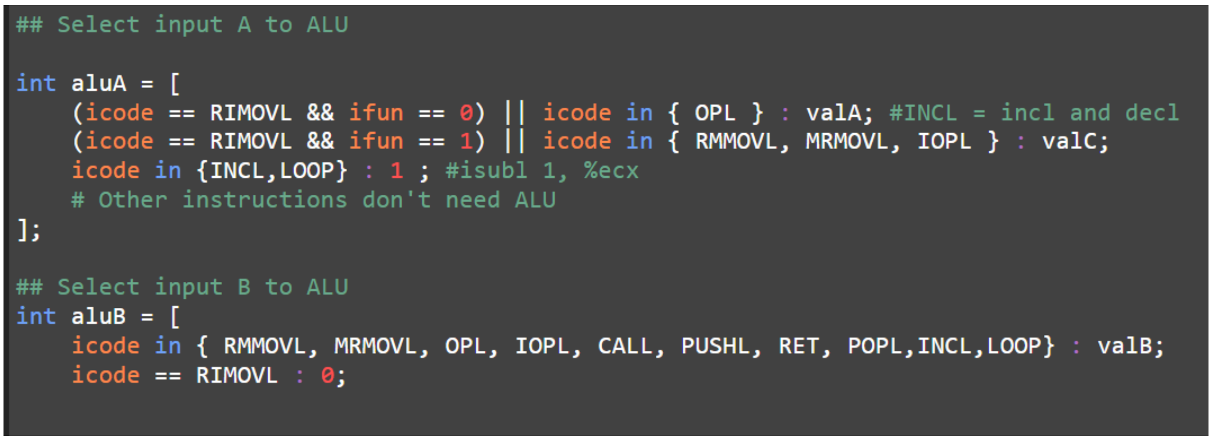
```
int srcB = [
    icode in { OPL, IOPL, RMMOVL, MRMOVL,INCL} : rB;
    icode in { PUSHL, POPL, CALL, RET } : RESP;
    icode == LOOP : RECX;
    1 : RNONE;  # Don't need register for reading
];
```



```
## What register should be used as the E destination?
int dstE = [
    icode in { RIMOVL, OPL, IOPL,INCL   } : rB;  # remove IRMOVL
    icode in { PUSHL, POPL, CALL, RET } : RESP;
    icode in {LOOP} :RECX;
    1 : RNONE;  # Don't need register for writing
];
```

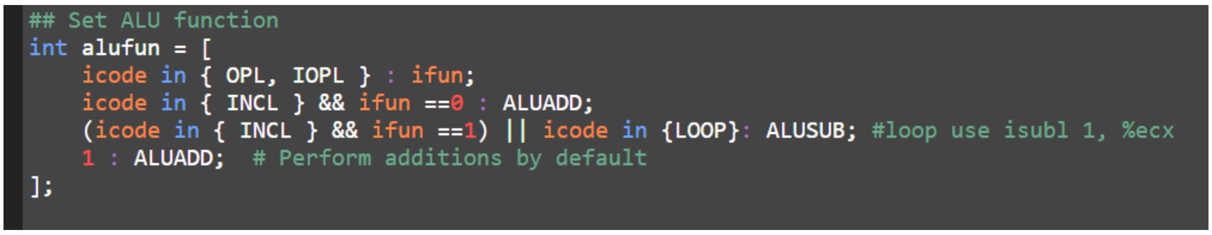
a. loop instruction implicitly uses isubl 1,%ecx so in this case ecx plays srcB and 1 should be a constant value used to decrement %ecx
b. the result will be stocked back into %ecx thus dstE: RECX when icode is loop

Step3: in Execute stage:



```
## Select input A to ALU

int aluA = [
    (icode == RIMOVL && ifun == 0) || icode in { OPL } : valA; #INCL = incl and decl
    (icode == RIMOVL && ifun == 1) || icode in { RMMOVL, MRMOVL, IOPL } : valC;
    icode in {INCL,LOOP} : 1 ; #isubl 1, %ecx
    # Other instructions don't need ALU
];

## Select input B to ALU
int aluB = [
    icode in { RMMOVL, MRMOVL, OPL, IOPL, CALL, PUSHL, RET, POPL,INCL,LOOP} : valB;
    icode == RIMOVL : 0;
```

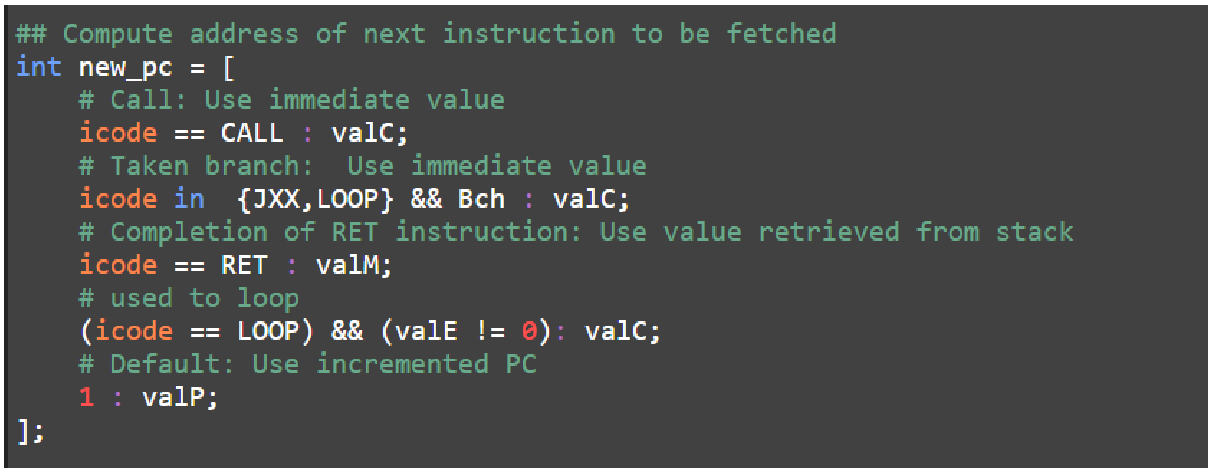
a. we recognise that to decrement ecx by 1, so we attribute 1 to aluA whenever the icode == LOOP.
b. aluB will take from srcB the value that will be decremented by aluA, in this case %ecx



```
## Set ALU function
int alufun = [
    icode in { OPL, IOPL } : ifun;
    icode in { INCL } && ifun ==0 : ALUADD;
    (icode in { INCL } && ifun ==1) || icode in {LOOP}: ALUSUB; #loop use isubl 1, %ecx
    1 : ALUADD;  # Perform additions by default
];
```

c. set the subtraction when icode == LOOP

Step 4: In updating pc stage:



```
## Compute address of next instruction to be fetched
int new_pc = [
    # Call: Use immediate value
    icode == CALL : valC;
    # Taken branch:  Use immediate value
    icode in  {JXX,LOOP} && Bch : valC;
    # Completion of RET instruction: Use value retrieved from stack
    icode == RET : valM;
    # used to loop
    (icode == LOOP) && (valE != 0): valC;
    # Default: Use incremented PC
    1 : valP;
];
```

a. we only concern about the logic icode==LOOP && valE!=0: valC, which means when this condition is true the pc will branch to the valC, in this case a label. If it is false, we continue with the next instruction

During testing, we ensured that the loop instruction decremented %ecx each time it was executed, until it reached 0, then executed halt. Everything worked as expected.

Exersice4

Question 1 In this part, we need to propose the encoding of ***loop**, **loope*** and ***loopne*** instructions, ensuring that the branch decision signal (Bch) is calculated correctly based on the value of the Z flag.

I. *We had already some pre-description of some instructions using icode==LOOP*
   a. **loop instruction:** This instruction decrements %ecx and branches to a label if %ecx is not zero. It doesn't modify the condition codes (Z, S, O), so the architecture can preserve these flags' state.
   b. **loope and loopne instructions**:
      - loope behaves like loop, but it only branches if %ecx is non-zero and the Z flag is set to 1.
      - loopne works similarly to loope, but it only branches if %ecx is non-zero and the Z flag is 0.

II. *And then when it comes to Bch signal:*

The Bch signal, which controls the branching decision, depends on the Z, S, and O flags. To ensure the correct behavior of loope and loopne, we need to set the ifun for each instruction appropriately:

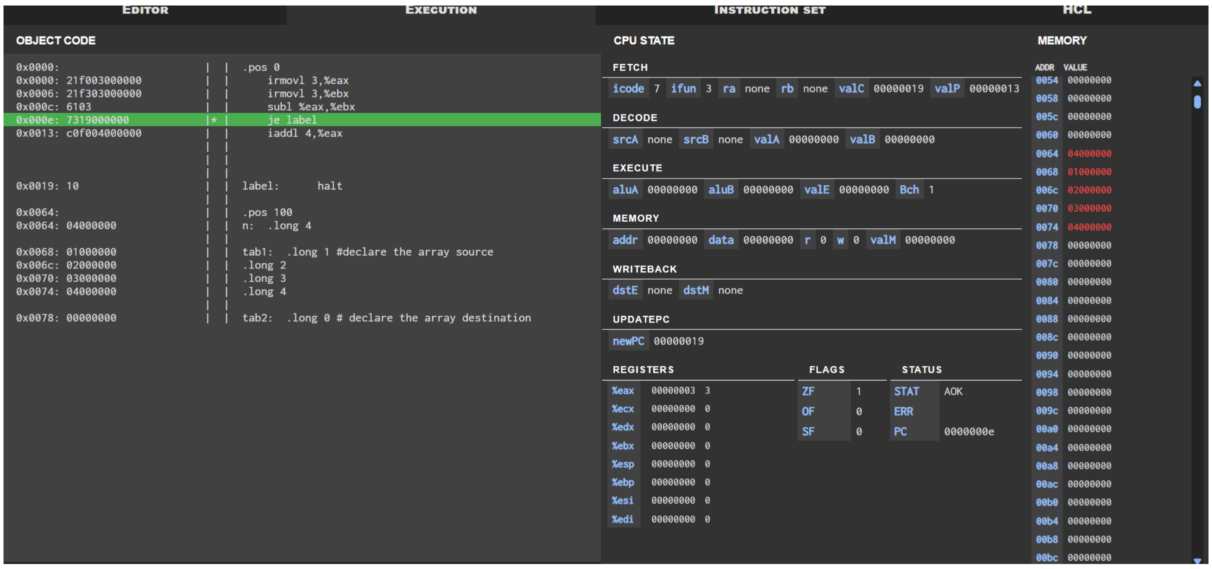
   a. **loop**: uses the existing ifun which is 0 in our case and should not be bothered by the flags, so to make it simple, we can just leave its HCL code like in exercice 3
   b. **loope and loopne instructions**:
      - loope: uses a new ifun that checks the Z flag (Z=1)
      - loopne: uses a different ifun that ensures branching only when the Z = 0.

The ifun will affect the computation of the Bch signal, enabling the instructions to behave as expected.

Question2 We have added support for loope and loopne instructions in the HCL code. The logic of branching was modified to incorporate conditions on the Z flag, while keeping the behavior of the loop instruction intact.

But the problems come at hand, how would we know what verifies Bch in order to enable the branching? Very easy, since **Bch signal** is the right combination between **ifun** and **flags**, we can try to guess the corresponding ifun that we need by reflecting JXX instruction.

Now we might want to look closer to their behavior:



OBJECT CODE

```
0x0000:                   |  |  .pos 0
0x0000: 21f003000000      |  |     irmovl 3,%eax
0x0006: 21f303000000      |  |     irmovl 3,%ebx
0x000c: 6103              |  |     subl %eax,%ebx
0x000e: 7319000000        |* |     je label
0x0013: c0f004000000      |  |     iaddl 4,%eax
                          |  |
                          |  |
                          |  |
0x0019: 10                |  |  label:     halt
                          |  |
0x0064:                   |  |  .pos 100
0x0064: 04000000          |  |  n:  .long 4
                          |  |
0x0068: 01000000          |  |  tab1:  .long 1 #declare the array source
0x006c: 02000000          |  |  .long 2
0x0070: 03000000          |  |  .long 3
0x0074: 04000000          |  |  .long 4
                          |  |
0x0078: 00000000          |  |  tab2:  .long 0 # declare the array destination
```

CPU STATE

```
FETCH
icode 7  ifun 3  ra none  rb none  valC 00000019  valP 00000013
DECODE
srcA none  srcB none  valA 00000000  valB 00000000
EXECUTE
aluA 00000000  aluB 00000000  valE 00000000  Bch 1
MEMORY
addr 00000000  data 00000000  r 0  w 0  valM 00000000
WRITEBACK
dstE none  dstM none
UPDATEPC
newPC 00000019

REGISTERS                  FLAGS        STATUS
%eax  00000003  3          ZF   1       STAT  AOK
%ecx  00000000  0          OF   0       ERR
%edx  00000000  0          SF   0       PC    0000000e
%ebx  00000000  0
%esp  00000000  0
%ebp  00000000  0
%esi  00000000  0
%edi  00000000  0
```

MEMORY

```
ADDR  VALUE
0054  00000000
0058  00000000
005c  00000000
0060  00000000
0064  04000000
0068  01000000
006c  02000000
0070  03000000
0074  04000000
0078  00000000
007c  00000000
0080  00000000
0084  00000000
0088  00000000
008c  00000000
0090  00000000
0094  00000000
0098  00000000
009c  00000000
00a0  00000000
00a4  00000000
00a8  00000000
00ac  00000000
00b0  00000000
00b4  00000000
00b8  00000000
00bc  00000000
```

a. we write this simple instruction set to test the Bch, when the arithmetic operation between %ebx and %eax is finished (%ebx=%ebx-%eax) the Z flag sets to 1 and with the corresponding ifun =3, Bch turns 1. Similarly, Bch also turns 1 when ifun=4 and Z flag=0

We want **loope** and **loopne** to perform similarly to **je** and **jne**, whose ifuns are respectively **3** and **4**

Effectively, the ifun of **loope** and **loopne** will be set respectively to **3** and **4** while keeping the icode == 14 in the instruction set

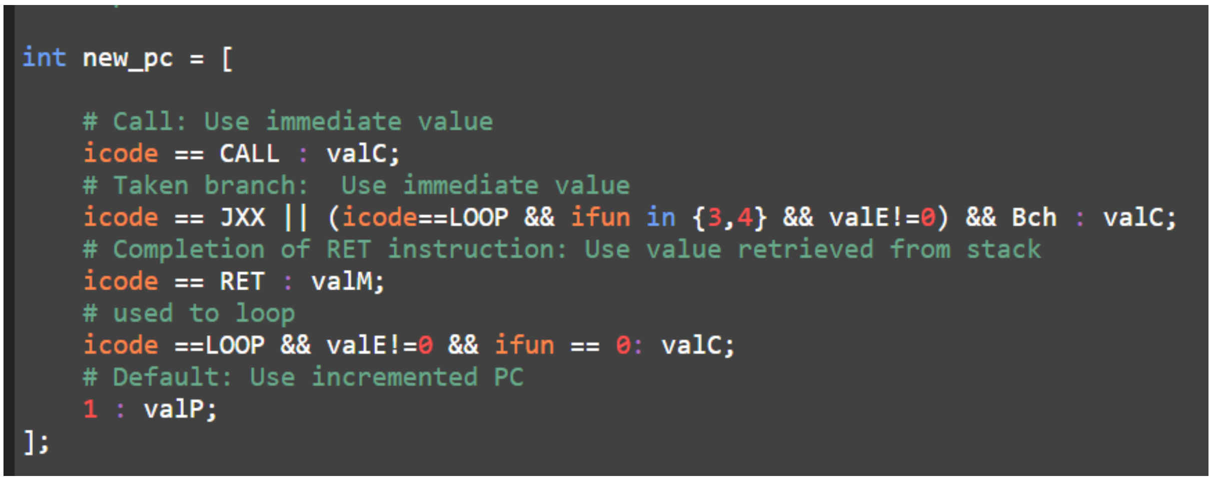
Here's how we updated the HCL code:

Step 1: in Execute stage

```
bool is_bch = icode in { JXX} || (icode ==LOOP && ifun in {3,4});
```

a. We add to is_bch one more condition stating that when icode == LOOP and ifun is either 3 or 4, Bch will be needed.

Step 2: in new_pc stage

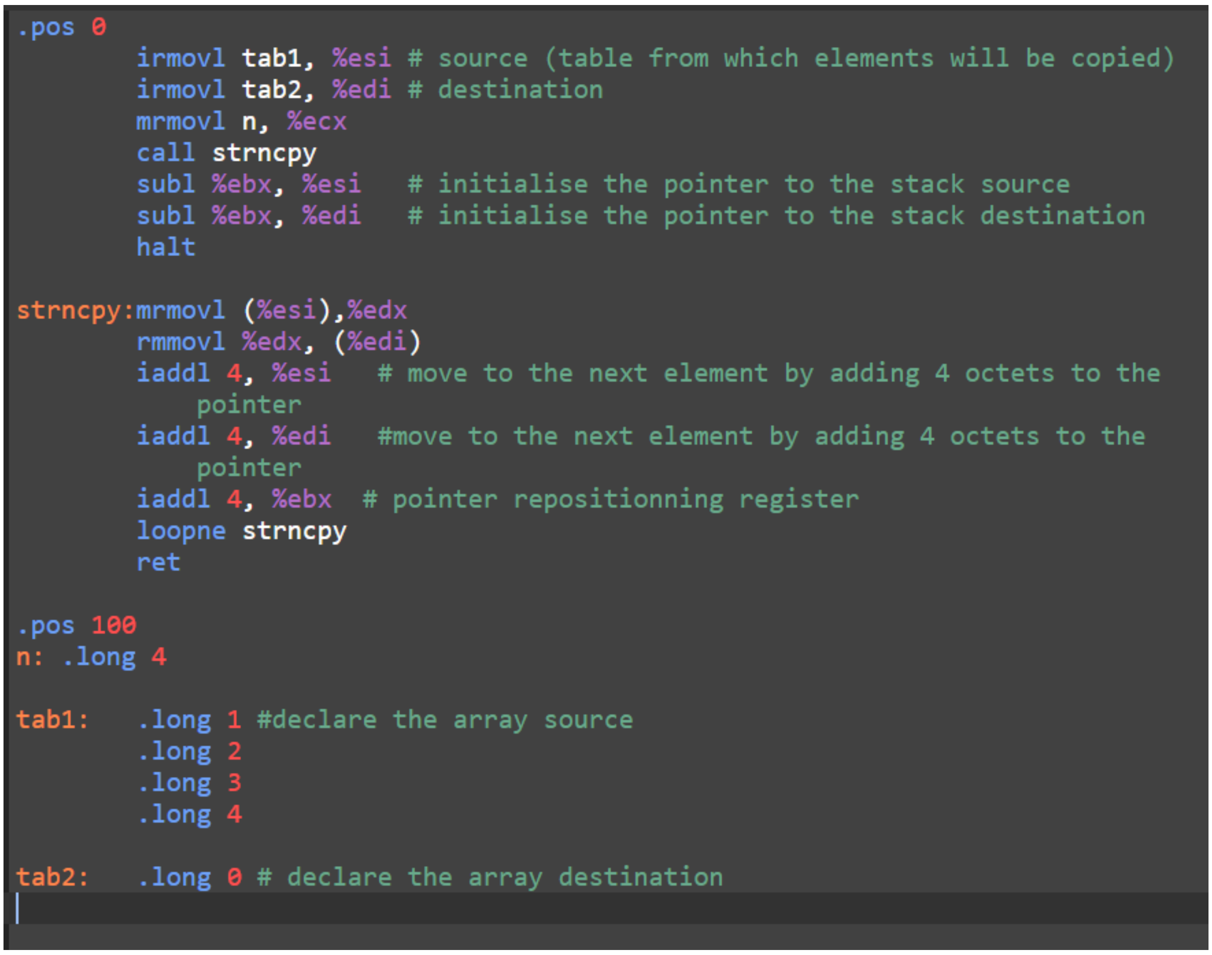


```
int new_pc = [

    # Call: Use immediate value
    icode == CALL : valC;
    # Taken branch:  Use immediate value
    icode == JXX || (icode==LOOP && ifun in {3,4} && valE!=0) && Bch : valC;
    # Completion of RET instruction: Use value retrieved from stack
    icode == RET : valM;
    # used to loop
    icode ==LOOP && valE!=0 && ifun == 0: valC;
    # Default: Use incremented PC
    1 : valP;
];
```

a. During this phase, we understand that the branching takes place only when the whole condition((icode==LOOP && ifun in {3,4} && valE!=0) && Bch) is true. We modified the new_pc calculation to consider these new instructions. Now, the decision to branch is not only based on the value of %ecx, but also on the Z flag, which changes depending on whether we are using loope or loopne.

** The rest of the HCL code includes necessary conditions to detect and handle these instructions accordingly.

Question3 For this question, we wrote a y86 version of the strncpy function using the loopne instruction. Since y86 handles only 32-bit words, this function operates on arrays of 32-bit integers rather than characters.
The goal of this function is to copy a given number of words from a source to a destination until we either reach a sentinel value (0) or the maximum number of words is copied.
Here’s the y86 code for the strncpy function using loopne:



```
.pos 0
        irmovl tab1, %esi # source (table from which elements will be copied)
        irmovl tab2, %edi # destination
        mrmovl n, %ecx
        call strncpy
        subl %ebx, %esi   # initialise the pointer to the stack source
        subl %ebx, %edi   # initialise the pointer to the stack destination
        halt

strncpy:mrmovl (%esi),%edx
        rmmovl %edx, (%edi)
        iaddl 4, %esi   # move to the next element by adding 4 octets to the
            pointer
        iaddl 4, %edi   #move to the next element by adding 4 octets to the
            pointer
        iaddl 4, %ebx  # pointer repositionning register
        loopne strncpy
        ret

.pos 100
n: .long 4

tab1:   .long 1 #declare the array source
        .long 2
        .long 3
        .long 4

tab2:   .long 0 # declare the array destination
```

Testing the function:
In the first case, the sentinel value 0 is found before the maximum number of words to copy is reached.
In the second case, the sentinel value is farther than the number of words to copy.
The test demonstrated that the loopne instruction works as intended, allowing words to be copied until either the sentinel value or the maximum word count is reached.