Código Final

En este punto el alumno debe de haber:

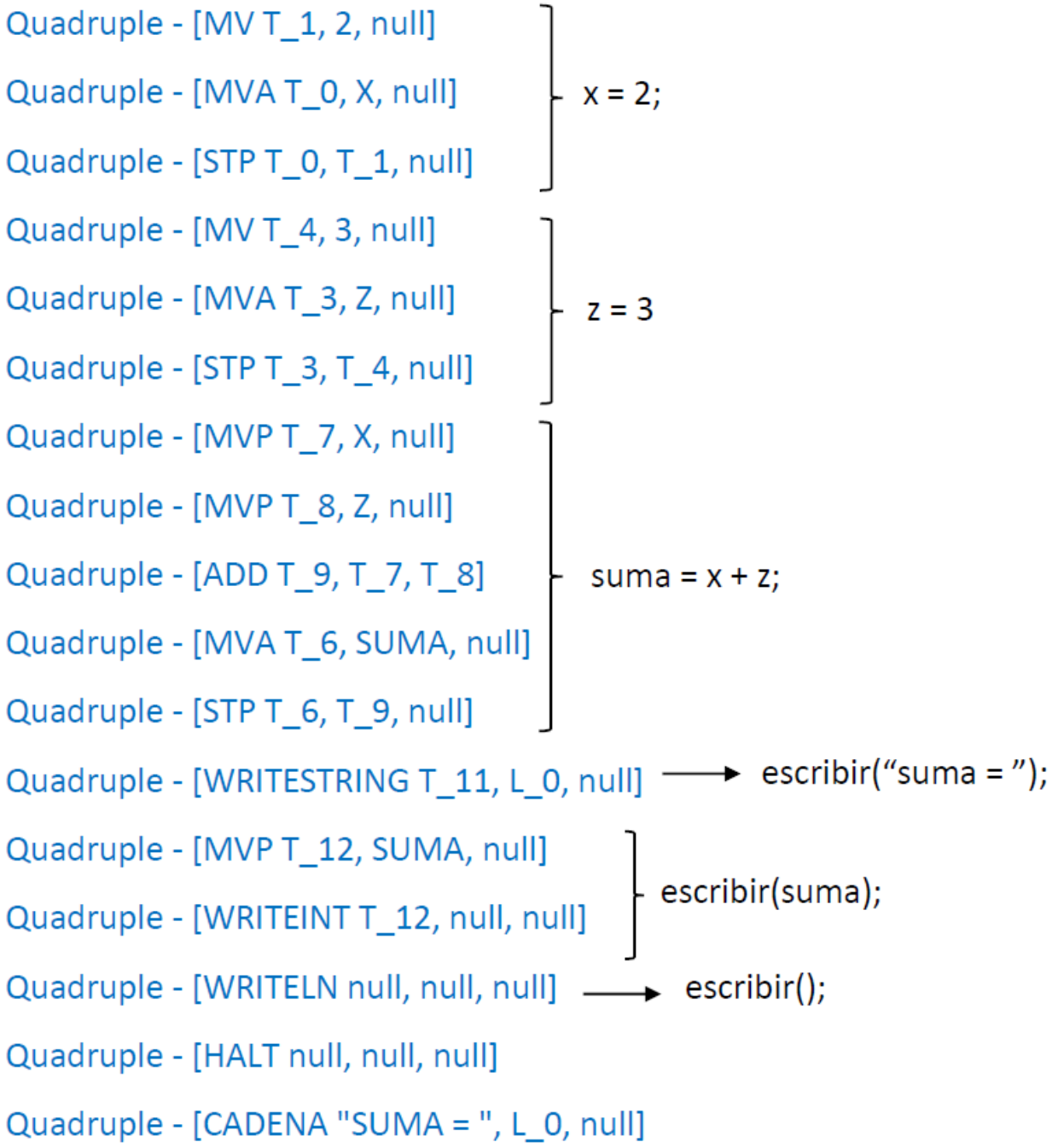
- Concluido el análisis semántico
  - Creación de los diferentes ámbitos
  - Declaración de tipos y símbolos en la tablas de sus correspondientes ámbitos
  - Comprobación de tipos:
    - Referencias y expresiones
    - Entre expresiones
    - Invocación de subprogramas
- Generar el código intermedio para las diferentes sentencias y expresiones.
  - Sentencias de asignación
  - Sentencias de control de flujo condicional(si-entonces-sino) e iterativo(para)
  - Sentencias de salida (escribe)

Queda excluido la generación de código intermedio para la invocación de subprogramas, tanto como sentencias (procedimientos) como en expresiones (funciones).

Programa ejemplo (test.muned):

```
programa test:
  variables
        suma, x, z : entero;
comienzo
   x =2;
   z =3;
   suma  = x + z;
   escribir ("suma = ");
   escribir(suma);
   escribir();
fin.
```

Código intermedio generado por el programa del ejemplo:



Es recomendable incluir una cuadrupla que indique el fin de la ejecución del programa principal, por ejemplo la cuadrupla **HALT** al final del bloque de sentencias del programa principal.

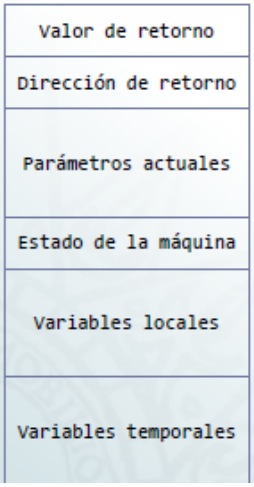
Registro de Activación (RA) es un espacio de memoria reservado, donde cada subprograma tendrá los datos relativos a su valor de retorno, dirección de retorno, parámetros actuales, variables, temporales, etc.

El lenguaje PL1UnedES se basa en un entorno de **ejecución basado en pila** para poder permitir la recursividad en los subprogramas, en concreto solo se dará soporte a la recursividad directa.

Entorno de ejecución estático

En los entornos de ejecución estáticos todas las activaciones de un mismo subprograma coinciden en memoria. Es decir, existe un único registro de activación por cada subprograma, incluido el programa principal (si es que éste no se incluye en la zona de datos estáticos). Esto permite conocer en tiempo de compilación cada direccionamiento pero imposibilita las activaciones recursivas

Diseño del registro de activación



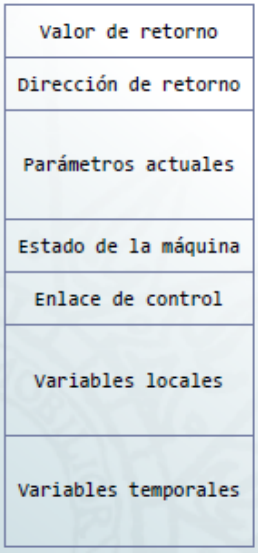
Nombre del campo	Escritor	Lector	Descripción
Valor de retorno	Llamado	Llamante	En funciones, almacena el valor devuelto por la función tras la invocación
Dirección de retorno	llamante	llamado	Se establece la dirección de código a la que debe saltar el llamado tras su invocación
Parámetros actuales	llamante	llamado	Proporciona los parámetros actuales
Estado de la máquina	Llamante	Llamado	Contiene el estado de la máquina (copia de los registros) antes de la llamada para restaurarlos tras su invocación
Variables locales	llamado	llamado	Contiene el valor de las variables locales
Variables temporales	llamado	llamado	Contiene el valor de las variables temporales

Referencia: Javier Vélez Reyes (Generación de código intermedio. Activación de subprogramas)

Entorno de ejecución basado en pila

En los entornos de ejecución dinámicos, cada activación distinta dispone de un registro de activación propio, gestionado a través de una pila. Esto complica la gestión de memoria ya que no todos los enlaces de datos son conocidos en tiempo de compilación pero permite la articulación de activaciones recursivas

Diseño del registro de activación



Nombre del campo	Escritor	Lector	Descripción
Valor de retorno	Llamado	Llamante	En funciones, almacena el valor devuelto por la función tras la invocación
Dirección de retorno	llamante	llamado	Se establece la dirección de código a la que debe saltar el llamado tras su invocación
Parámetros actuales	llamante	llamado	Proporciona los parámetros actuales
Estado de la máquina	Llamante	Llamado	Contiene el estado de la máquina (copia de los registros) antes de la llamada para restaurarlos tras su invocación
Enlace de control	Llamante	Llamado	Contiene un puntero al Registro de activación del subprograma llamante
Variables locales	llamado	llamado	Contiene el valor de las variables locales
Variables temporales	llamado	llamado	Contiene el valor de las variables temporales

Referencia: Javier Vélez Reyes (Generación de código intermedio. Activación de subprogramas)

Referencias no locales

En programas con estructura de bloques anidados (Pascal, Modula-2, Ada), las referencias no locales se refieren a variables locales a algún ámbito accesible desde los RA de los hijos según la regla de anidamiento del ámbito más cercano. En A11, la referencia n es no local ya que está definida localmente en el ámbito A, accesible desde éste

Diseño del registro de activación

Nombre del campo	Escritor	Lector	Descripción
Valor de retorno	Llamado	Llamante	En funciones, almacena el valor devuelto por la función tras la invocación
Dirección de retorno	llamante	llamado	Se establece la dirección de código a la que debe saltar el llamado tras su invocación
Parámetros actuales	llamante	llamado	Proporciona los parámetros actuales
Estado de la máquina	Llamante	Llamado	Contiene el estado de la máquina (copia de los registros) antes de la llamada para restaurarlos tras su invocación
Enlace de control	Llamante	Llamado	Contiene un puntero al Registro de activación del subprograma llamante
Enlace de acceso	Llamante	Llamado	Contiene un puntero al Registro de activación del subprograma anidante (padre)
Variables locales	llamado	llamado	Contiene el valor de las variables locales
Variables temporales	llamado	llamado	Contiene el valor de las variables temporales

Referencia: Javier Vélez Reyes (Generación de código intermedio. Activación de subprogramas)

Registro de Activación

El registro índice **IX** lo usaremos como puntero de marco, y la pila de la memoria empezara en las posiciones superiores e ira decrementándose.

#0[.IX]	Valor de Retorno
#-1[.IX]	Enlace de Control
#-2[.IX]	Estado Máquina
#-3[.IX]	Enlace de Acceso
#-4[.IX]	Parámetro Actuales
#-4 - nP[.IX]	Dirección de Retorno
#-5 - nP[.IX]	Variables locales
#-5 - np - nV[.IX]	Temporales

nP = número de parámetros
nV = número de variables locales

Nota: Al tratarse de un Registro de Activación genérico tanto para el programa principal como para un procedimiento o una función, algunos campos de este Registro de Activación pueden usarse o no. También se podría diseñar un Registro de Activación específico para cada propósito.

ENS2001

Se recomienda al alumno que lea el documento Herramientas -> ENS2001-Windows/DOS/Linux -> Manual del Usuario.pdf

La Máquina Virtual que simula la aplicación posee las siguientes características:

- **Procesador** con ancho de palabra de 16 bits.
- **Memoria** de 64 K palabras (de 16 bits cada una). Por tanto, el direccionamiento es de 16 bits, coincidiendo con el ancho de palabra, desde la dirección 0 a la 65535 (FFFFh).
- **Banco de Registros**. Todos ellos son de 16 bits.
- **PC** (Contador de Programa): Indica la posición en memoria de la siguiente instrucción que se va a ejecutar.
- **SP** (Puntero de Pila): Indica la posición de memoria donde se encuentra la cima libre de la pila.
- **SR** (Registro de Estado): Almacena el conjunto de los biestables de estado.
- **IX**, **IY** (Registros Índices): Se emplean para efectuar direccionamientos relativos.
- **A** (Acumulador): Almacena el resultado de las operaciones aritméticas y lógicas de dos operandos.
- **R0**..**R9** (Registros de Propósito General): Son registros cuyo uso decidirá el programador en cada momento.

Modos de direccionamiento

- Direccionamiento inmediato
  - MOVE #67, /1000 MOVE #67,.R1
- Direccionamiento directo a registro
  - SUB .IX, #8 SUB .R1,#8
- Direccionamiento directo a memoria
  - MOVE #67, /1000 INC /1000
- Direccionamiento indirecto
  - MOVE #-8[.IX], [.R1] DEC [.R1]
- Direccionamiento Relativo a registro índice
  - MOVE #-8[.IX], [.R1] DEC #-8[.IX]
- Direccionamiento Relativo a contador del programa
  - BR $3 BR /bucle

Asignación de posiciones estáticas de memoria para las variables y temporales del programa principal sin usar Registro de Activación:

- En el Axioma de lenguaje vamos a recuperar el ámbito del programa principal, ámbito de nivel 0 y vamos a ir asignado las posiciones de memoria que correspondientes a las variables y temporales.
- Para ello crearemos una variable llamada por ejemplo, **direccionEstatica**, la cual inicializaremos con un dirección de memoria, por ejemplo MAX_ADDRESS = 65535 C:\..\ArquitecturaPLII-cursoXX\src\compiler\code\ExecutionEnvironmentEns2001.java) e iremos asignando posiciones en orden decreciente de la memoria.
- Empezaremos asignando la posición de memoria para las variables locales del programa principal y ajustamos el valor de la variable direccionEstatica con el tamaño de la variable local (1 para variables locales de tipo primitivo y para las variables de tipo definido por usuario, según su tamaño).
- Asignaremos las posiciones de memoria para los temporales del programa principal (1 posición por cada temporal).
- Generamos el código intermedio para el programa principal creando las cuádruplas para inicializar las variables del programa principal a cero.
    - VARGLOBAL var, 0 (inicializar las variables a cero)
- Finalmente, todo el código intermedio generado se lo pasamos al código final.

```
program ::=
{: syntaxErrorManager.syntaxInfo ("Starting parsing...");
:}
 axiom:ax
{:
//Asignación de posiciones de memoria para las variables globales y los temporales
List<ScopeIF> scopes = scopeManager.getAllScopes();
for (ScopeIF scope: scopes) {
        int direccionEstatica = CodigoFinal.getMemorySize();     // MAX_ADDRESS = 65535
        List<SymbolIF> simbolos = scope.getSymbolTable().getSymbols();
        for (SymbolIF simbolo: simbolos) {
                if (simbolo instanceof SymbolVariable) {  //Comprobar si el símbolo es una variable
//Guardamos la dirección del variable en SymbolVariable
//C:\..\ArquitecturaPLII-cursoXX\src\compiler\semantic\symbol\SymbolVariable.java
                        ((SymbolVariable)simbolo).setAddress(direccionEstatica);
                        //Actualizamos el valor de direccionEstatica para el próximo símbolo
                        direccionEstatica = direccionEstatica - simbolo.getType().getSize();
                }
        }
```

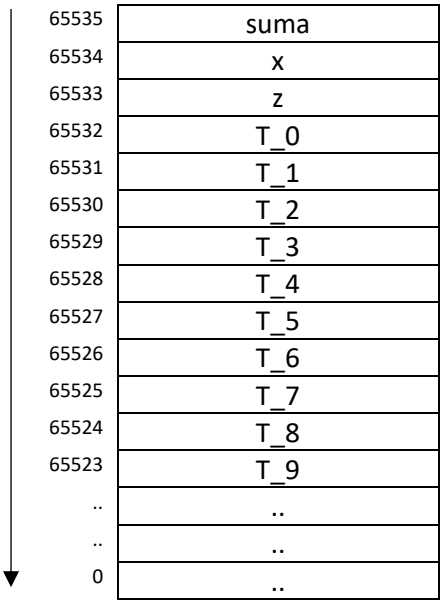
```
		//al igual que se hicimos con las variables lo hacemos con los temporales
		List<TemporalIF> temporales = scope.getTemporalTable ().getTemporals();
		for (TemporalIF t: temporales) {
			if (t instanceof Temporal) {
//Guardamos la dirección del temporal en Temporal.java
// C:\..\ArquitecturaPLII-cursoXX\src\compiler\intermediate\Temporal.java
				((Temporal)t).setAddress(direccionEstatica);
				direccionEstatica = direccionEstatica - ((Temporal)t).getSize();
			}
		}
}
//Generación de código intermedio para iniciar el programa principal e inicializar las variables globales
for (ScopeIF scope: scopes) {
		IntermediateCodeBuilder cb = new IntermediateCodeBuilder(scope);
		if (scope.getLevel () == 0) {
			List<SymbolIF> simbolos = scope.getSymbolTable ().getSymbols();
			for (SymbolIF simbolo: simbolos) {
				if (simbolo instanceof SymbolVariable) {  //Comprobar si el simbolo es una variable
					int direccion = ((SymbolVariable)simbolo).getAddress();
					Variable var = new Variable(simbolo.getName(), simbolo.getScope());
					// Se inicializa las variables del programa principal a 0
					cb.addQuadruple("VARGLOBAL", var, 0);
				}
			}
		}
		cb.addQuadruples(ax.getIntermediateCode());
		ax.setIntermediateCode(cb.create());
}

// No modificar esta estructura, aunque se pueden añadir más acciones semánticas
//printamos el código intermedio generado
System.out.println("Codigo intermedio en el AXIOMA: " + ax.getIntermediateCode());
List intermediateCode = ax.getIntermediateCode ();
finalCodeFactory.setEnvironment(CompilerContext.getExecutionEnvironment());
finalCodeFactory.create (intermediateCode);
// En caso de no comentarse las sentencias anteriores puede generar una excepcion
// en las llamadas a cupTest si el compilador no está completo. Esto es debido a que
// aún no se tendrá implementada la generación de código intermedio ni final.
// Para la entrega final deberán descomentarse y usarse.

syntaxErrorManager.syntaxInfo ("Parsing process ended.");
:};
```

Quadruple - [VARGLOBAL SUMA, 0, null]
Quadruple - [VARGLOBAL, X, 0, null]
Quadruple - [VARGLOBAL, Z, 0, null]
Quadruple - [MV T_1, 2, null]
Quadruple - [MVA T_0, X, null]
Quadruple - [STP T_0, T_1, null]
Quadruple - [MV T_4, 3, null]
Quadruple - [MVA T_3, Z, null]
Quadruple - [STP T_3, T_4, null]
Quadruple - [MVP T_7, X, null]
Quadruple - [MVP T_8, Z, null]
Quadruple - [ADD T_9, T_7, T_8]
Quadruple - [MVA T_6, SUMA, null]
Quadruple - [STP T_6, T_9, null]
Quadruple - [WRITESTRING T_11, L_0, null]
Quadruple - [MVP T_12, SUMA, null]
Quadruple - [WRITEINT T_12, null, null]
Quadruple - [WRITELN null, null, null]
Quadruple - [HALT null, null, null]
Quadruple - [CADENA "SUMA = ", L_0, null]

Posiciones de memoria asignadas a las variables y temporales, por lo que para acceder a una dirección de memoria de una variable o temporal usaremos un direccionamiento directo a memoria.



Asignación de posición dentro del R.A. de las variables y temporales del programa principal usando un Registro de Activación.

- En el Axioma de lenguaje vamos a recuperar todos los ámbitos que se han creado e iremos asignado las posiciones de memoria que corresponde a las variables, parámetros y temporales de cada ámbito.
- La variable **direccionRA** la inicializaremos a 4 que incluye las posiciones reservadas para el Valor de Retorno, Enlace de Control, Estado de la Máquina y Enlace de Acceso del Registro de Activación propuesto.
- Asignaremos las posiciones de memoria para las variables locales del programa principal.
- Asignaremos las posiciones de memoria para los temporales del programa principal.
- Generamos el código intermedio para el programa principal creando las cuádruplas
    - STARTGLOBAL (inicializa el R.A. del programa principal)
    - VARGLOBAL var, 0 (inicializar las variables a cero)
    - PUNTEROGLOBAL temp, tamaño (posiciona el .SP según el tamaño del R.A. global)
- Finalmente, todo el código intermedio generado se lo pasamos al código final.

```
program ::=
{: syntaxErrorManager.syntaxInfo ("Starting parsing...");
:}
 axiom:ax
{:
//Asignación de posiciones de memoria dentro del R.A. para las variables globales y los temporales
List<ScopeIF> scopes = scopeManager.getAllScopes();
for (ScopeIF scope: scopes) {
        int direccionRA = 4;      //posiciones reservadas dentro del R.A.(Valor de Retorno + Enlace de Control +
                                  //Estado Máquina + Enlace de Acceso)
        List<SymbolIF> simbolos = scope.getSymbolTable().getSymbols();
        for (SymbolIF simbolo: simbolos) {
                if (simbolo instanceof SymbolVariable) {  //Comprobar si el símbolo es una variable
//Guardamos la dirección del variable en SymbolVariable
//C:\..\ArquitecturaPLII-cursoXX\src\compiler\semantic\symbol\SymbolVariable.java
                        ((SymbolVariable)simbolo).setAddress(direccionRA);
                        //Actualizamos el valor de direccionRA para el próximo símbolo
                        direccionRA = direccionRA + simbolo.getType().getSize();
                }
        }
```

```
		//al igual que se hicimos con las variables lo hacemos con los temporales
		List<TemporalIF> temporales = scope.getTemporalTable ().getTemporals();
		for (TemporalIF t: temporales) {
			if (t instanceof Temporal) {
//Guardamos la dirección del temporal en Temporal.java
// C:\..\ArquitecturaPLII-cursoXX\src\compiler\intermediate\Temporal.java
				((Temporal)t).setAddress(direccionRA);
				direccionRA = direccionRA + ((Temporal)t).getSize();
			}
		}
}
//Generación de código intermedio para iniciar el programa principal e inicializar las variables globales
for (ScopeIF scope: scopes) {
		IntermediateCodeBuilder cb = new IntermediateCodeBuilder(scope);
		TemporalFactory tF = new TemporalFactory(scope);
		TemporalIF temp = tF.create();
		int tamanoRA;
		tamanoRA = scope.getSymbolTable().getSize() + scope.getTemporalTable().getSize() + 4;
		if (scope.getLevel () == 0) {
			cb.addQuadruple("STARTGLOBAL");  //Prepara el R.A. global
			List<SymbolIF> simbolos = scope.getSymbolTable ().getSymbols();
			for (SymbolIF simbolo: simbolos) {
				if (simbolo instanceof SymbolVariable) {  //Comprobar si el simbolo es una variable
					int direccion = ((SymbolVariable)simbolo).getAddress();
					Variable var = new Variable(simbolo.getName(), simbolo.getScope());
					// introduce la variable en el R.A. GLOBAL inicializadas a 0
					cb.addQuadruple("VARGLOBAL", var, 0);
				}
			}
			// posiciona el .SP según el tamaño del R.A. del programa principal
			cb.addQuadruple("PUNTEROGLOBAL", temp, tamanoRA);
		}
		cb.addQuadruples(ax.getIntermediateCode());
		ax.setIntermediateCode(cb.create());
}

// No modificar esta estructura, aunque se pueden añadir más acciones semánticas
//printamos el código intermedio generado
System.out.println("Codigo intermedio en el AXIOMA: " + ax.getIntermediateCode());
List intermediateCode = ax.getIntermediateCode ();
finalCodeFactory.setEnvironment(CompilerContext.getExecutionEnvironment());
finalCodeFactory.create (intermediateCode);
// En caso de no comentarse las sentencias anteriores puede generar una excepcion
// en las llamadas a cupTest si el compilador no está completo. Esto es debido a que
// aún no se tendrá implementada la generación de código intermedio ni final.
// Para la entrega final deberán descomentarse y usarse.

syntaxErrorManager.syntaxInfo ("Parsing process ended.");
:};
```

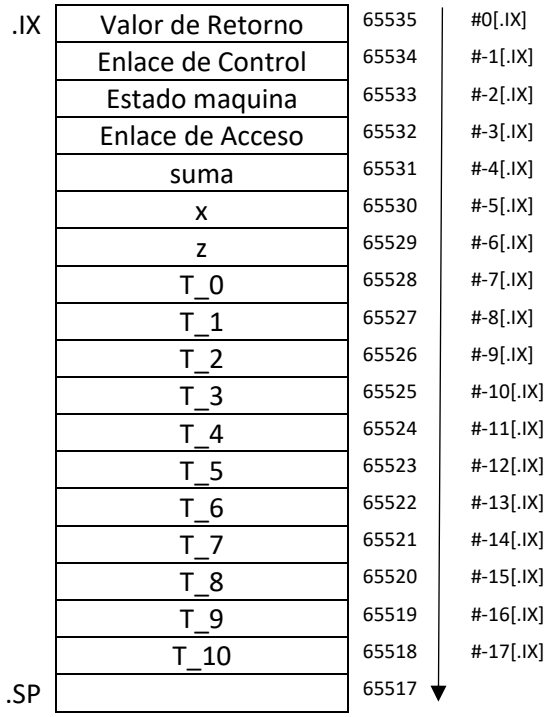
Quadruple - [STARTGLOBAL null, null, null],
Quadruple - [VARGLOBAL SUMA, 0, null],
Quadruple - [VARGLOBAL X, 0, null],
Quadruple - [VARGLOBAL Z, 0, null],
Quadruple - [PUNTEROGLOBAL T_10, 18, null],
Quadruple - [MV T_0, 2, null],
Quadruple - [MVA T_1, X, null],
Quadruple - [STP T_1, T_0, null],
Quadruple - [MV T_2, 3, null],
Quadruple - [MVA T_3, Z, null],
Quadruple - [STP T_3, T_2, null],
Quadruple - [MVP T_4, X, null],
Quadruple - [MVP T_5, Z, null],
Quadruple - [ADD T_6, T_4, T_5],
Quadruple - [MVA T_7, SUMA, null],
Quadruple - [STP T_7, T_6, null],
Quadruple - [WRITESTRING T_8, L_0, null],
Quadruple - [MVP T_9, SUMA, null],
Quadruple - [WRITEINT T_9, null, null],
Quadruple - [WRITELN null, null, null],
Quadruple - [HALT null, null, null],
Quadruple - [CADENA "SUMA = ", L_0, null]

Dentro del Registro de Activación vamos asignando posiciones de memoria a los valores fijos del registro (Valor de retorno, enlace de control,…) variables y temporales. Para acceder a las posiciones de memoria que ocupa el registro de Activación lo haremos a través de direccionamiento relativo al registro índice IX.

Registro de Activación

0	Valor de Retorno
1	Enlace de Control
2	Estado maquina
3	Enlace de Acceso
4	suma
5	x
6	z
7	T_0
8	T_1
9	T_2
10	T_3
11	T_4
12	T_5
13	T_6
14	T_7
15	T_8
16	T_9
17	T_10

Pila de la memoria



C:\..\ArquitecturaPLII-cursoXX\src\compiler\semantic\symbol\SymbolVariable.java
Incluir una variable con sus métodos get/set para la dirección asignada en memoria.

```
private int direccion;

public int getAddress(){
        return this.direccion;
}
public void setAddress(int valor){
        this.direccion=valor;
}
```

Para los temporales podemos usar la variable address
C:\..\ ArquitecturaPLII-cursoXX\src\compiler\intermediate\Temporal.java

Tipos primitivos lógico

Se debe de crear un convenio donde los tipos primitivos lógicos TRUE y FALSE se transformen en un valor entero, por ejemplo TRUE igual a 1 y FALSE igual a 0. Esta transformación se puede hacer bien al generar el código intermedio o bien al traducir el código intermedio en código final.

```
exprexion ::= TRUE
{:
        Expresion ex = new Expresion();
        //Código intermedio
        ScopeIF scope = scopeManager.getCurrentScope();
        TemporalFactory tf = new TemporalFactory(scope);
        IntermediateCodeBuilder cb = new IntermediateCodeBuilder(scope);
        TemporalIF temp = tf.create();
        cb.addQuadruple("MV", temp, new Value(1));
        ex.setTemporal(temp);
        ex.setIntermediateCode(cb.create());
        RESULT = ex;
:}

exprexion ::= FALSE:n
{:
        Expresion ex = new Expresion();

        //Código intermedio
        ScopeIF scope = scopeManager.getCurrentScope();
        TemporalFactory tf = new TemporalFactory(scope);
        IntermediateCodeBuilder cb = new IntermediateCodeBuilder(scope);
        TemporalIF temp = tf.create();
        cb.addQuadruple("MV", temp, new Value(n.getLexema()));
        ex.setTemporal(temp);
        ex.setIntermediateCode(cb.create());
        RESULT = ex;
:}
```

Convertir cuadruplas en código final

C:\..\ArquitecturaPLII-cursoXX\src\compiler\code\ExecutionEnvironmentEns2001.java

```
/**
    * Translate a quadruple into a set of final code instructions.
    * @param cuadruple The quadruple to be translated.
    * @return a quadruple into a set of final code instructions.
    */
   @Override
   public final String translate (QuadrupleIF quadruple)
   {
         //TODO: Student work

         //resta dos operando y guarda el valor en el registro acumulador que luego mueve al resultado.
         if(quadruple.getOperation().equals("ADD")) {
               StringBuffer b = new StringBuffer();
               String o1 = operacion(quadruple.getFirstOperand());
               String o2 = operacion(quadruple.getSecondOperand());
               String r = operacion(quadruple.getResult());
               b.append(";" + quadruple.toString() + "\n");  //generar cuadrupla como un comentario (opcional)
               b.append("ADD" + o1 + ", " + o2 + "\n");
               b.append("MOVE " + ".A " + ", " + r);
               return b.toString();
         }

      return quadruple.toString();
   }
```

Usando direcciones estáticas

Quadruple - [ADD T_6, T_4, T_5] → ADD /65528, /65527
MOVE .A , /65526

Usando R.A.

Quadruple - [ADD T_6, T_4, T_5] → ADD #-11[.IX], #-12[.IX]
MOVE .A , #-13[.IX]

```
if(quadruple.getOperation().equals("MV")) {  //mueve el valor de una variable o temporal a otro
         StringBuffer b = new StringBuffer();
         String o1 = operacion(quadruple.getFirstOperand());
         String r = operacion(quadruple.getResult());
         b.append(";" + quadruple.toString() + "\n");  //generar cuadrupla como un comentario (opcional)
         b.append("MOVE " + o1 + ", " + r);
         return b.toString();
}
```

Usando direcciones estáticas

Quadruple - [MV T_2, 3, null] → MOVE #3, /65530

Usando R.A.

Quadruple - [MV T_2, 3, null] → MOVE #3, #-9[.IX]

Sin R.A. con asignaciones estáticas de posiciones de memoria.

```
private String operacion (OperandIF o) {
	if(o instanceof Variable) {
		return "/" + ((Variable)o).getAddress();
	}

	if(o instanceof Value){
		return "#" + ((Value)o).getValue();
	}

	if(o instanceof Temporal){
		return "/" + ((Temporal)o).getAddress();
	}

	if(o instanceof Label){
		return ((Label)o).getName();
	}

	return null;

}
```

Con R.A. y desplazamiento relativo al registro índice IX

```
private String operacion (OperandIF o) {

	if(o instanceof Variable) {
		return "#-" + ((Variable)o).getAddress() + "[.IX]";
	}

	if(o instanceof Value){
		return "#" + ((Value)o).getValue();
	}

	if(o instanceof Temporal){
		return "#-" + ((Temporal)o).getAddress() + "[.IX]";
	}

	if(o instanceof Label){
		return ((Label)o).getName();
	}

	return null;

}
```

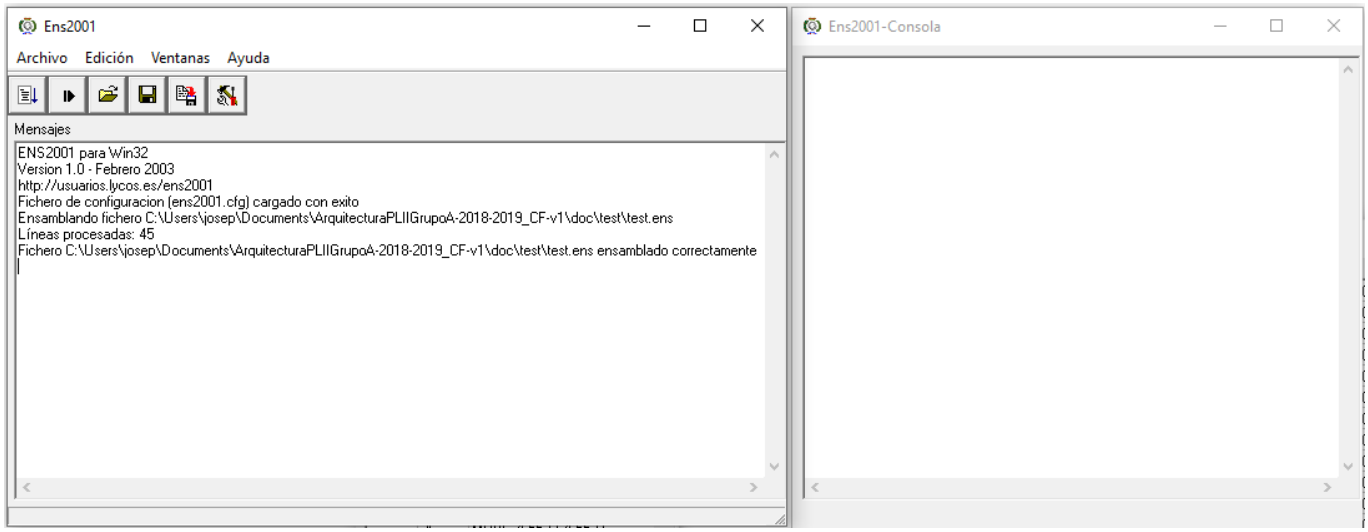
Nota : (Variable)o).getAddress(), se debe de crear un método getAddres() en C:\..\ ArquitecturaPLII-cursoXX\src\compiler\intermediate\Variable.java que devuelva el valor de la posición de memoria o posición dentro del R.A. del symbolVariable.java

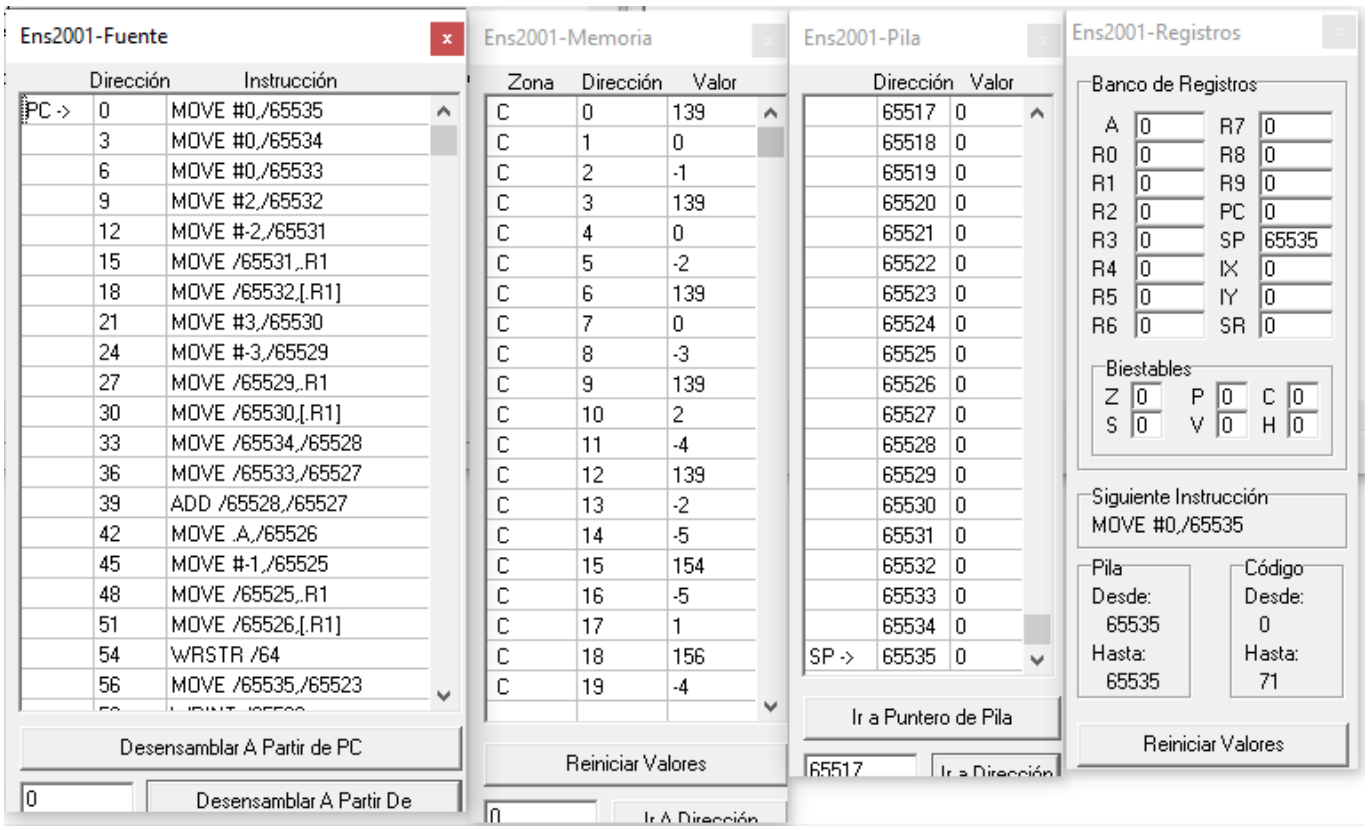
Código ensamblador (ens2001) generado por el ejemplo, usando direcciones estáticas de memoria:

```
;Quadruple - [VARGLOBAL SUMA, 0, null]
MOVE #0, /65535
;Quadruple - [VARGLOBAL X, 0, null]
MOVE #0, /65534
;Quadruple - [VARGLOBAL Z, 0, null]
MOVE #0, /65533
;Quadruple - [MV T_0, 2, null]
MOVE #2, /65532
;Quadruple - [MVA T_1, X, null]
MOVE #65534, /65531
;Quadruple - [STP T_1, T_0, null]
MOVE /65531, .R1
MOVE /65532, [.R1]
;Quadruple - [MV T_2, 3, null]
MOVE #3, /65530
;Quadruple - [MVA T_3, Z, null]
MOVE #65533, /65529
;Quadruple - [STP T_3, T_2, null]
MOVE /65529, .R1
MOVE /65530, [.R1]
;Quadruple - [MVP T_4, X, null]
MOVE /65534, /65528
;Quadruple - [MVP T_5, Z, null]
MOVE /65533, /65527
;Quadruple - [ADD T_6, T_4, T_5]
ADD /65528, /65527
MOVE .A , /65526
;Quadruple - [MVA T_7, SUMA, null]
MOVE #65535, /65525
;Quadruple - [STP T_7, T_6, null]
MOVE /65525, .R1
MOVE /65526, [.R1]
;Quadruple - [WRITESTRING T_8, L_0, null]
WRSTR /L_0
;Quadruple - [MVP T_9, SUMA, null]
MOVE /65535, /65523
;Quadruple - [WRITEINT T_9, null, null]
WRINT /65523
;Quadruple - [WRITELN null, null, null]
WRCHAR #10
;Quadruple - [HALT null, null, null]
HALT
;Quadruple - [CADENA "SUMA = ", L_0, null]
L_0 : DATA "SUMA = "
```

Evolución de la pila y los registros ens2001

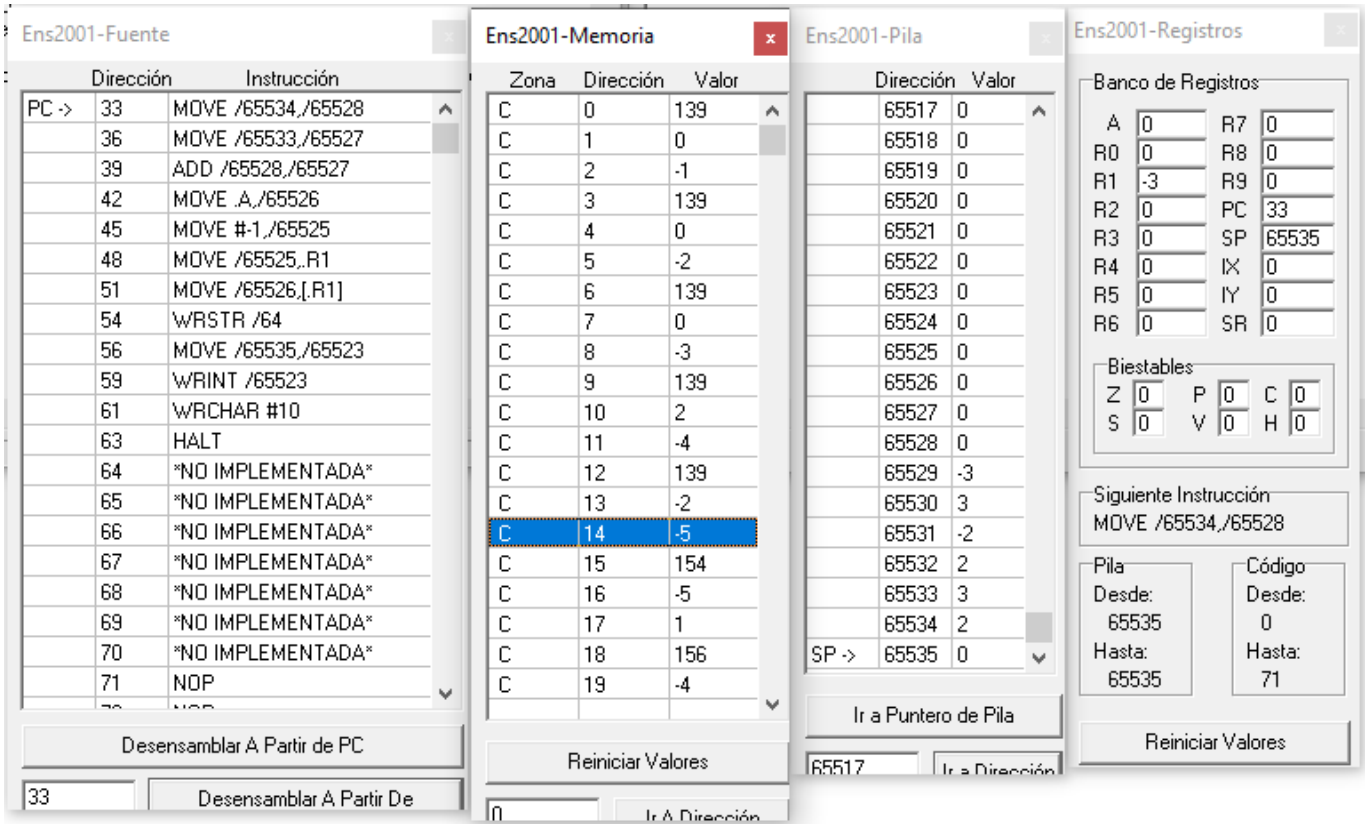
Archivo -> Abrir y Ensamblar -> C:\..\ArquitecturaPLII-cursoXX\doc\test\test.ens





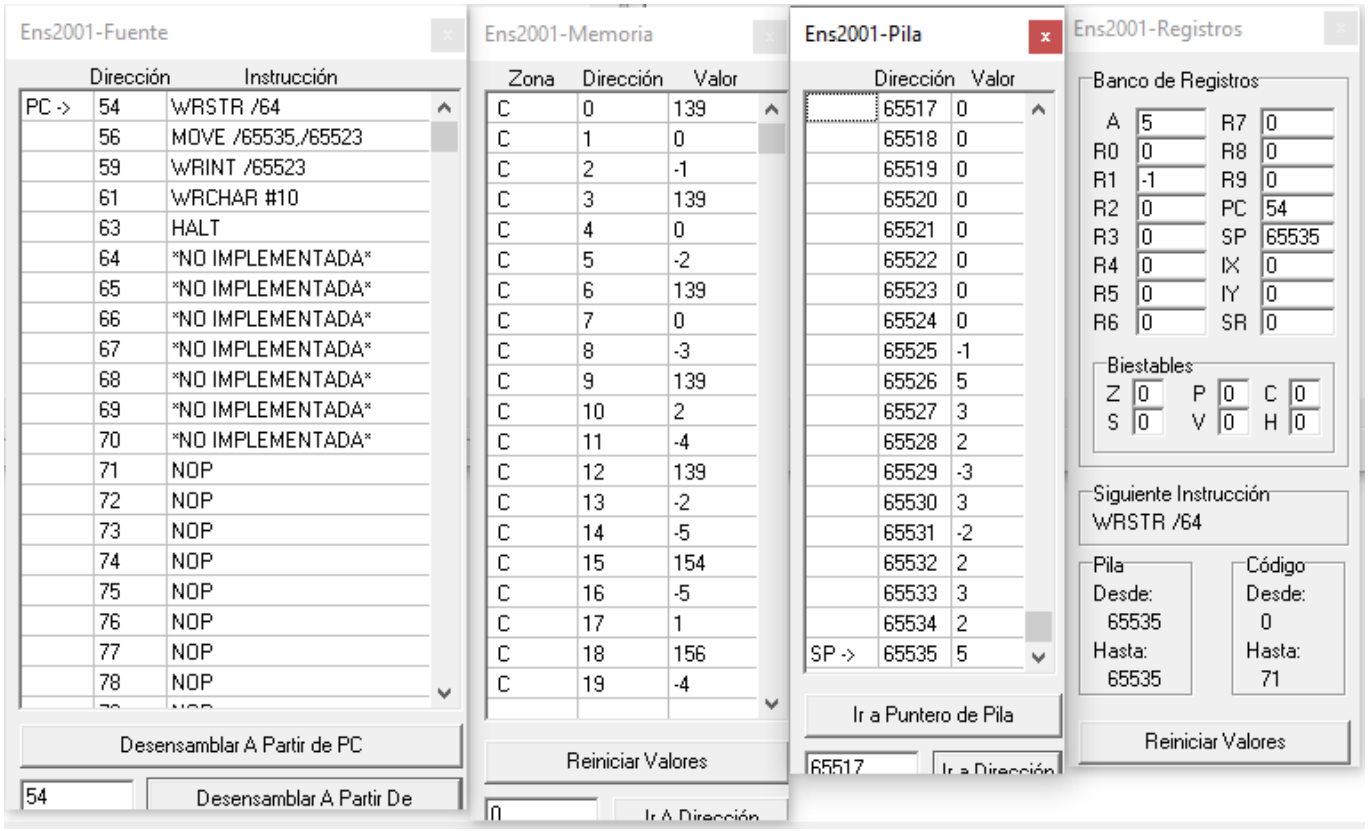
Después de las instrucciones de asignar valores a las variables “x” y “z”
x = 2;
z = 3;

MOVE #0, /65535
MOVE #0, /65534
MOVE #0, /65533
MOVE #2, /65532
MOVE #65534, /65531
MOVE /65531, .R1
MOVE /65532, [.R1]
MOVE #3, /65530
MOVE #65533, /65529
MOVE /65529, .R1
MOVE /65530, [.R1]



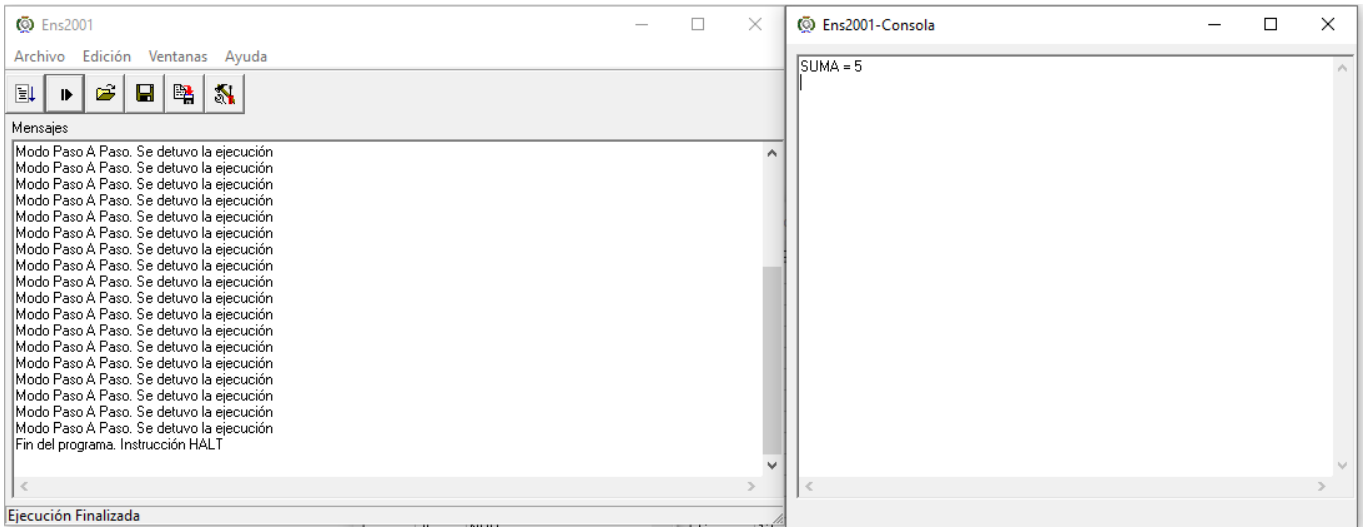
Después de ejecutar las instrucciones para asignar el valor de x+z a la variable suma:
suma = x + z;

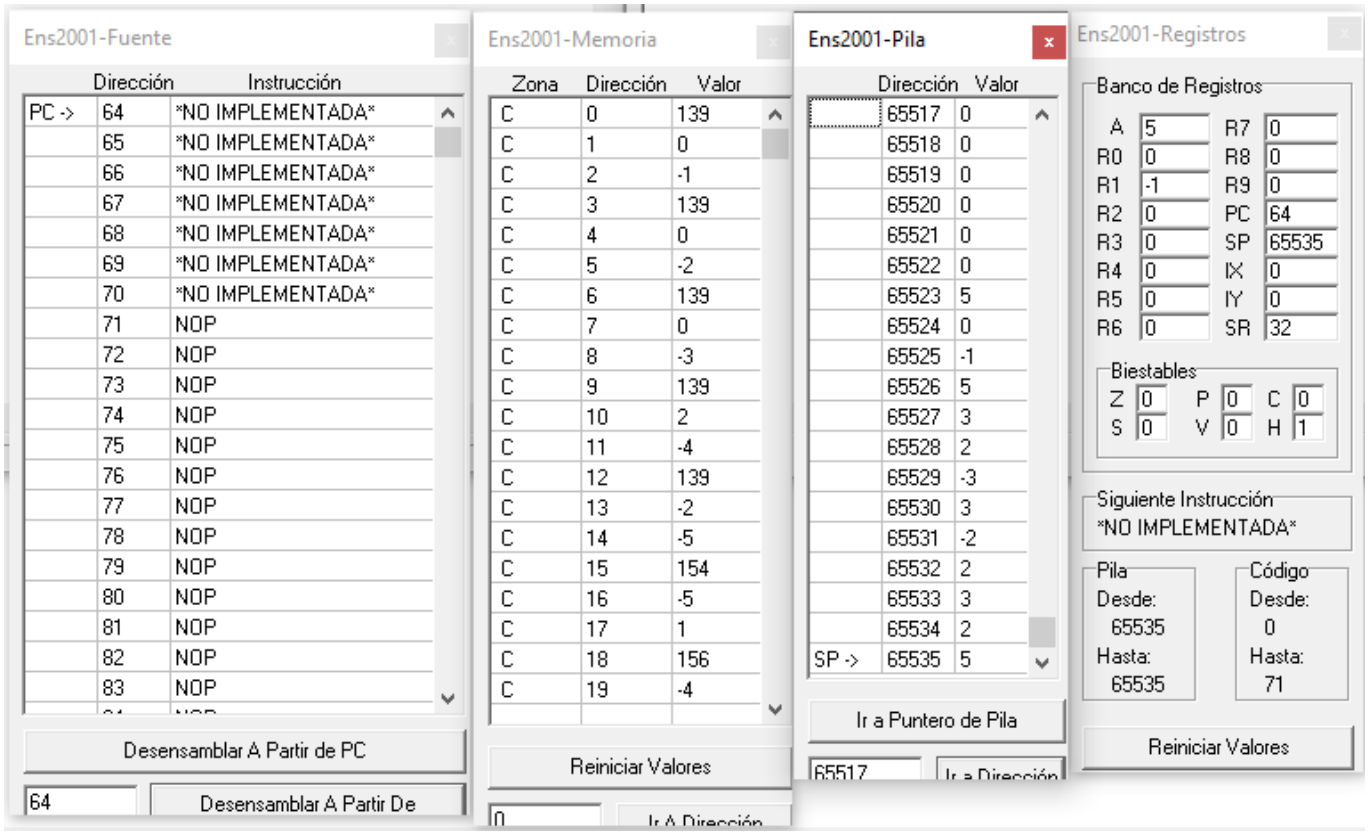
MOVE /65534, /65528
MOVE /65533, /65527
ADD /65528, /65527
MOVE .A , /65526
MOVE #65535, /65525
MOVE /65525, .R1
MOVE /65526, [.R1]



Una vez finalizada la ejecución del programa principal

```
WRSTR /L_0
MOVE /65535, /65523
WRINT /65523
WRCHAR #10
HALT
L_0 : DATA "SUMA = "
```



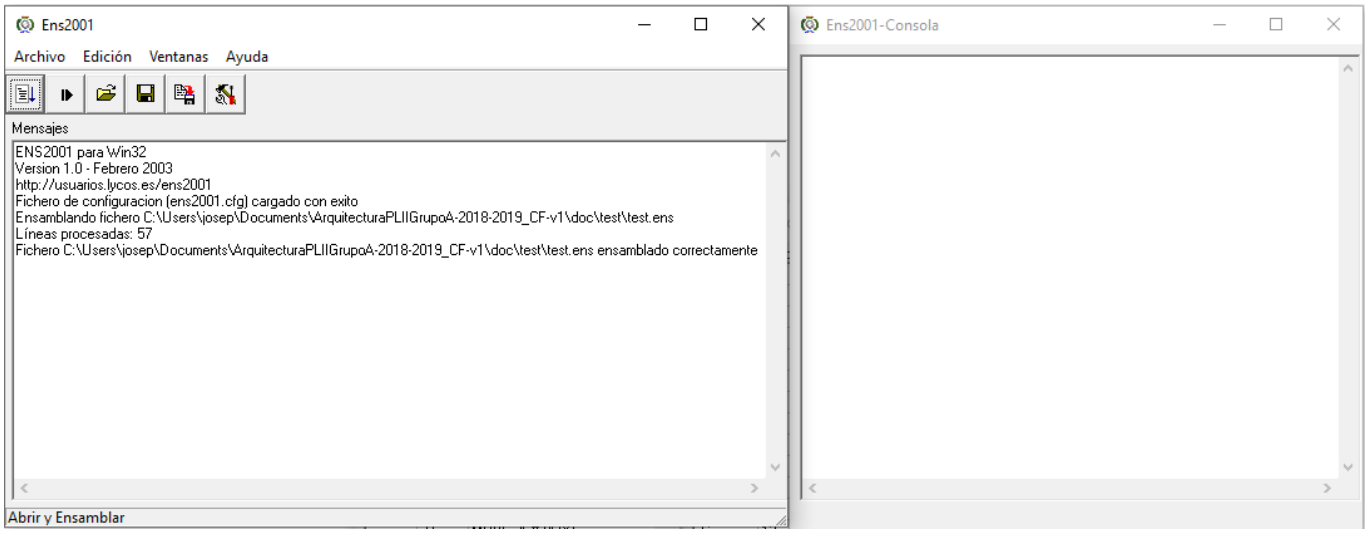


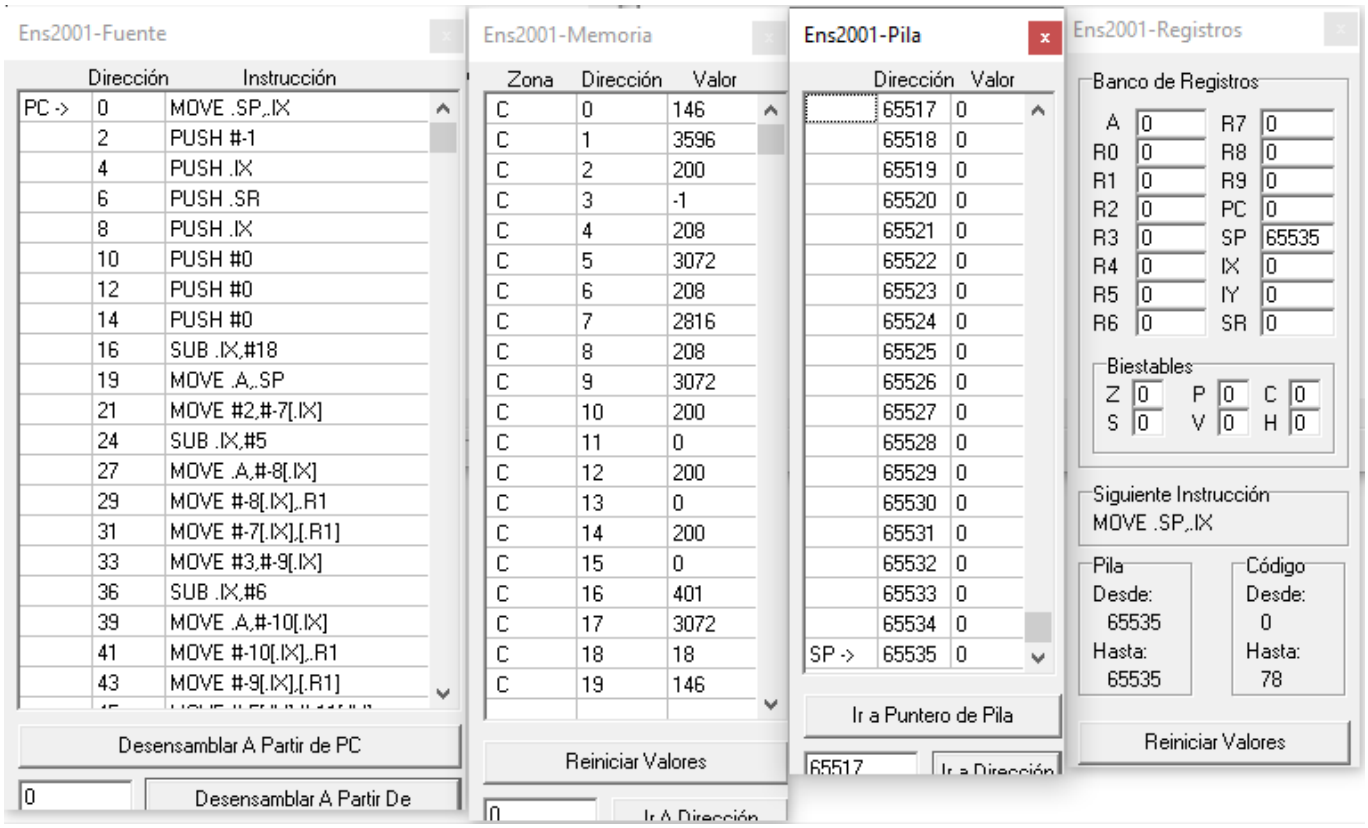
Código ensamblador (ens2001) generado por el ejemplo, usando un Registro de Activación:

```
;Quadruple - [STARTGLOBAL null, null, null]
MOVE .SP, .IX
PUSH #-1
PUSH .IX
PUSH .SR
PUSH .IX
;Quadruple - [VARGLOBAL SUMA, 0, null]
PUSH #0
;Quadruple - [VARGLOBAL X, 0, null]
PUSH #0
;Quadruple - [VARGLOBAL Z, 0, null]
PUSH #0
;Quadruple - [PUNTEROGLOBAL T_10, 18, null]
SUB .IX, #18
MOVE .A, .SP
;Quadruple - [MV T_0, 2, null]
MOVE #2, #-7[.IX]
;Quadruple - [MVA T_1, X, null]
SUB .IX, #5
MOVE .A, #-8[.IX]
;Quadruple - [STP T_1, T_0, null]
MOVE #-8[.IX], .R1
MOVE #-7[.IX], [.R1]
;Quadruple - [MV T_2, 3, null]
MOVE #3, #-9[.IX]
;Quadruple - [MVA T_3, Z, null]
SUB .IX, #6
MOVE .A, #-10[.IX]
;Quadruple - [STP T_3, T_2, null]
MOVE #-10[.IX], .R1
MOVE #-9[.IX], [.R1]
;Quadruple - [MVP T_4, X, null]
MOVE #-5[.IX], #-11[.IX]
;Quadruple - [MVP T_5, Z, null]
MOVE #-6[.IX], #-12[.IX]
;Quadruple - [ADD T_6, T_4, T_5]
ADD #-11[.IX], #-12[.IX]
MOVE .A , #-13[.IX]
;Quadruple - [MVA T_7, SUMA, null]
SUB .IX, #4
MOVE .A, #-14[.IX]
;Quadruple - [STP T_7, T_6, null]
MOVE #-14[.IX], .R1
MOVE #-13[.IX], [.R1]
;Quadruple - [WRITESTRING T_8, L_0, null]
WRSTR /L_0
;Quadruple - [MVP T_9, SUMA, null]
MOVE #-4[.IX], #-16[.IX]
;Quadruple - [WRITEINT T_9, null, null]
```

```
WRINT #-16[.IX]
;Quadruple - [WRITELN null, null, null]
WRCHAR #10
;Quadruple - [HALT null, null, null]
HALT
;Quadruple - [CADENA "SUMA = ", L_0, null]
L_0 : DATA "SUMA = "
```

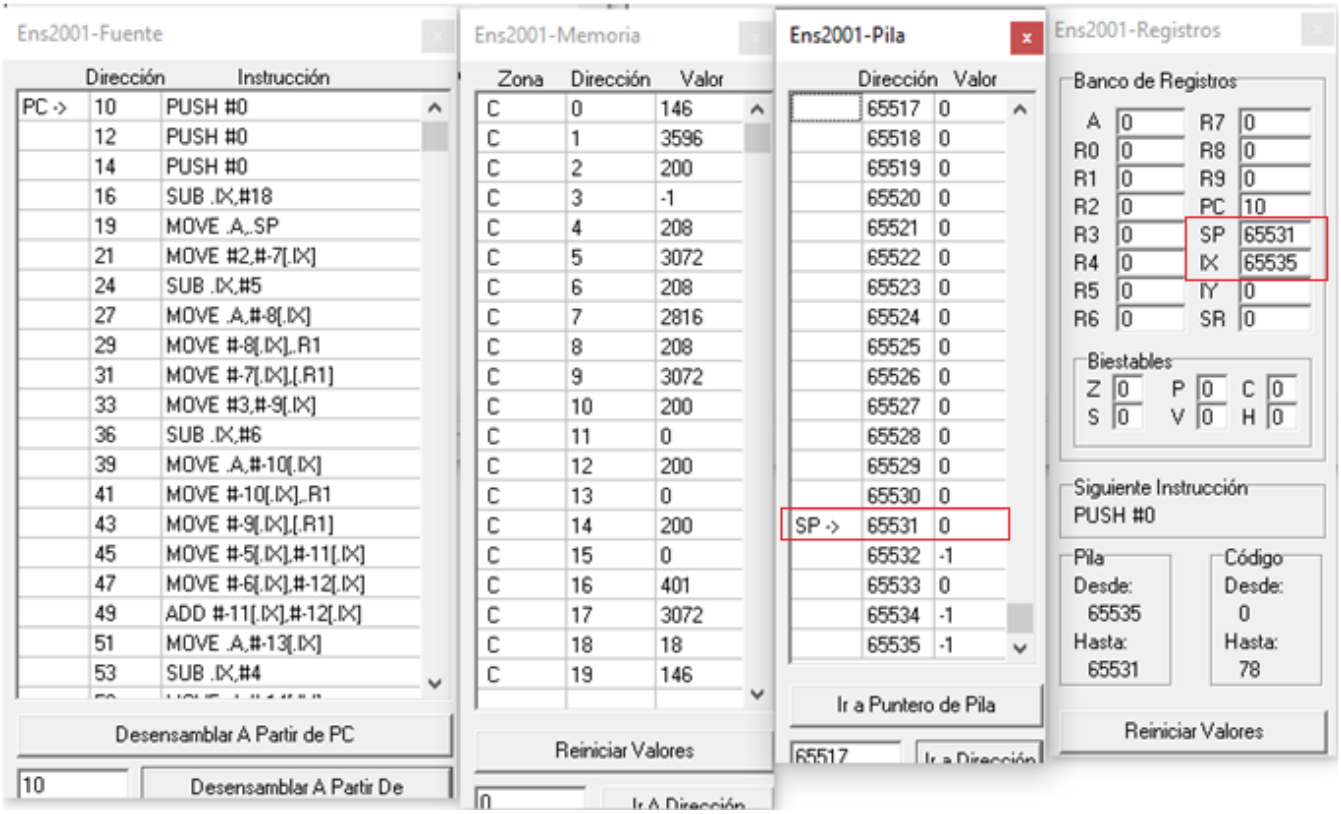
Archivo -> Abrir y Ensamblar -> C:\..\ArquitecturaPLII-cursoXX\doc\test\test.ens





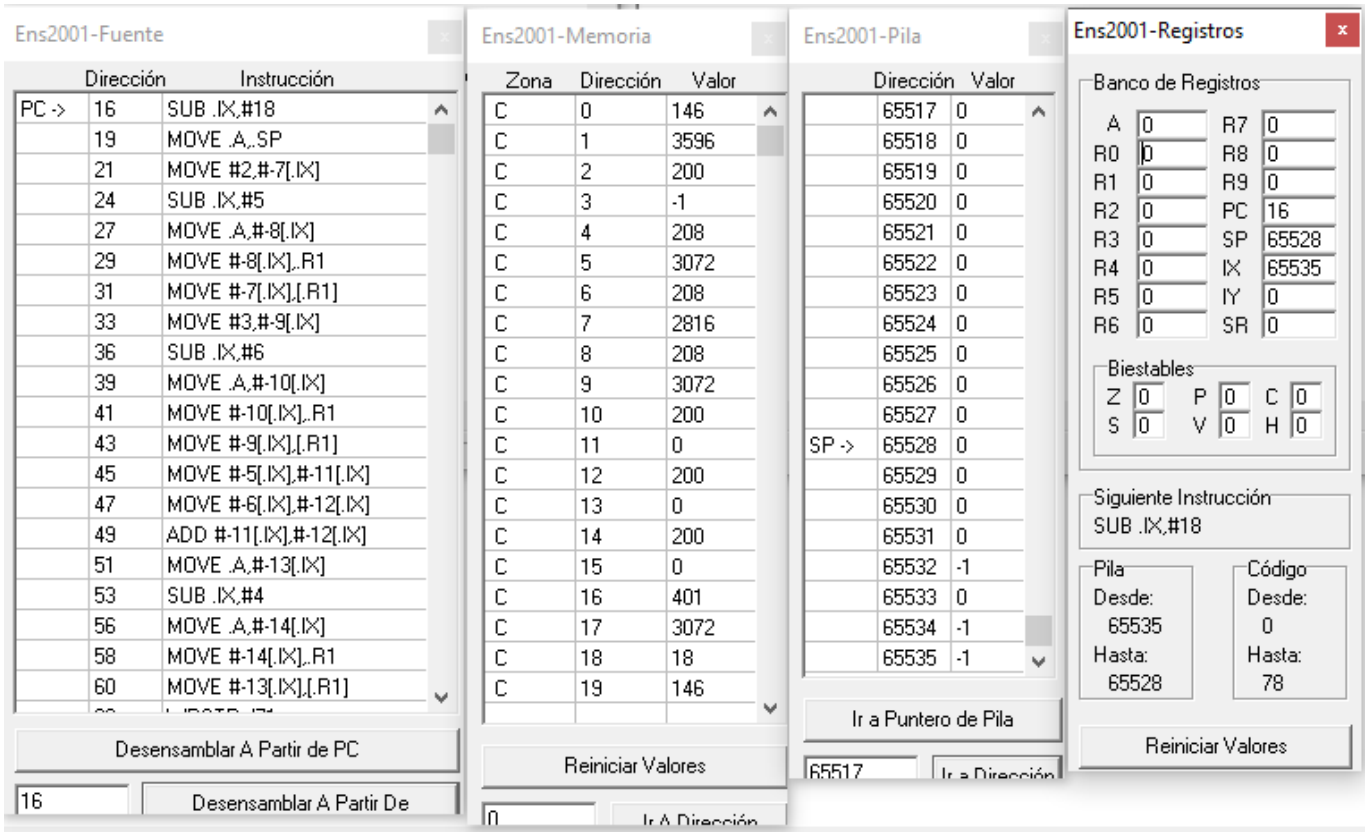
Después de ejecutar de las instrucciones de inicializar el Registro de Activación de programa principal.

MOVE .SP, .IX
PUSH #-1
PUSH .IX
PUSH .SR
PUSH .IX



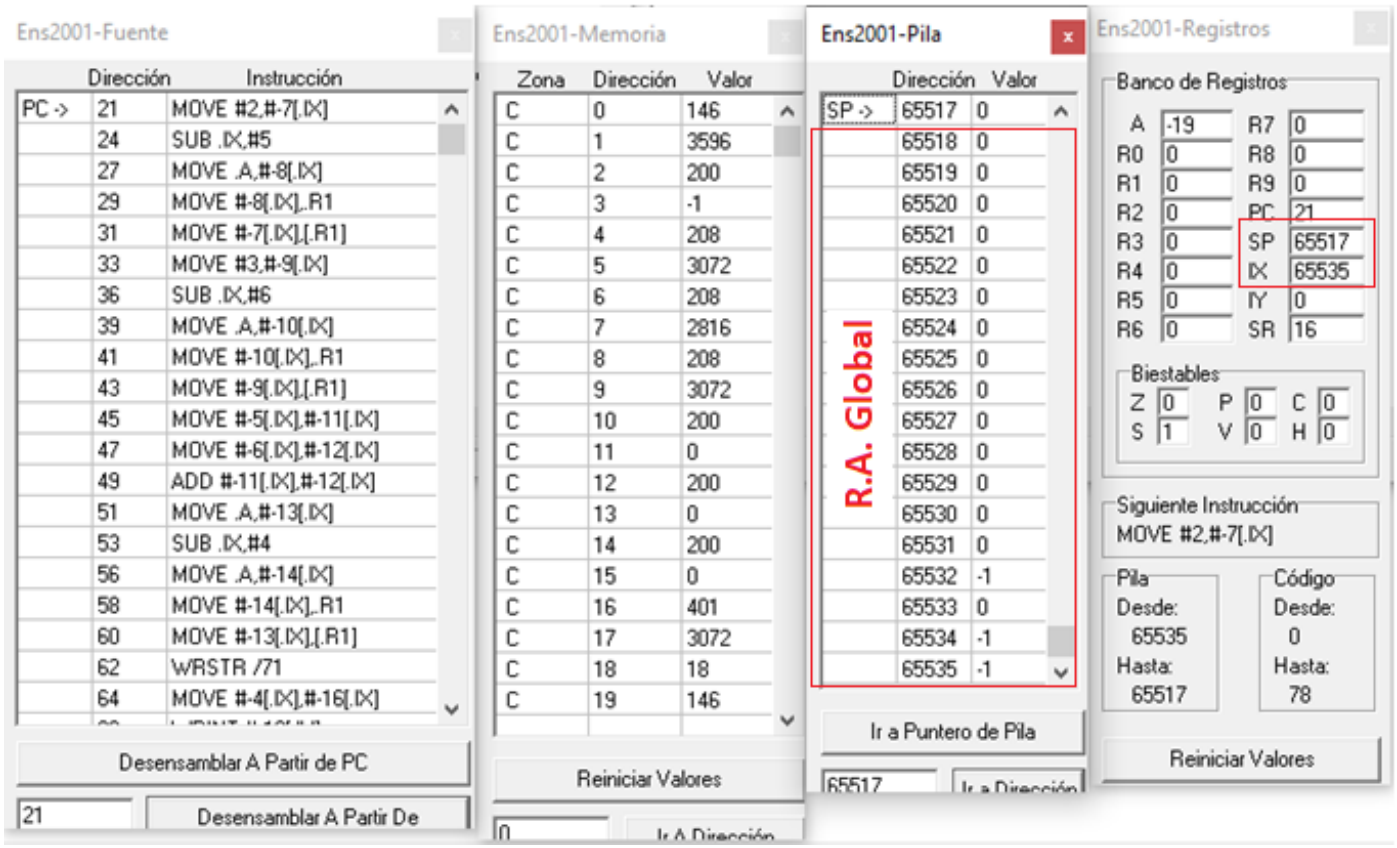
Después de ejecutar las instrucciones de inicialización de las variables del programa principal:

PUSH #0
PUSH #0
PUSH #0



Después de ejecutar las instrucciones de posicionar el puntero de pila reservando el espacio del registro de activación del programa principal:

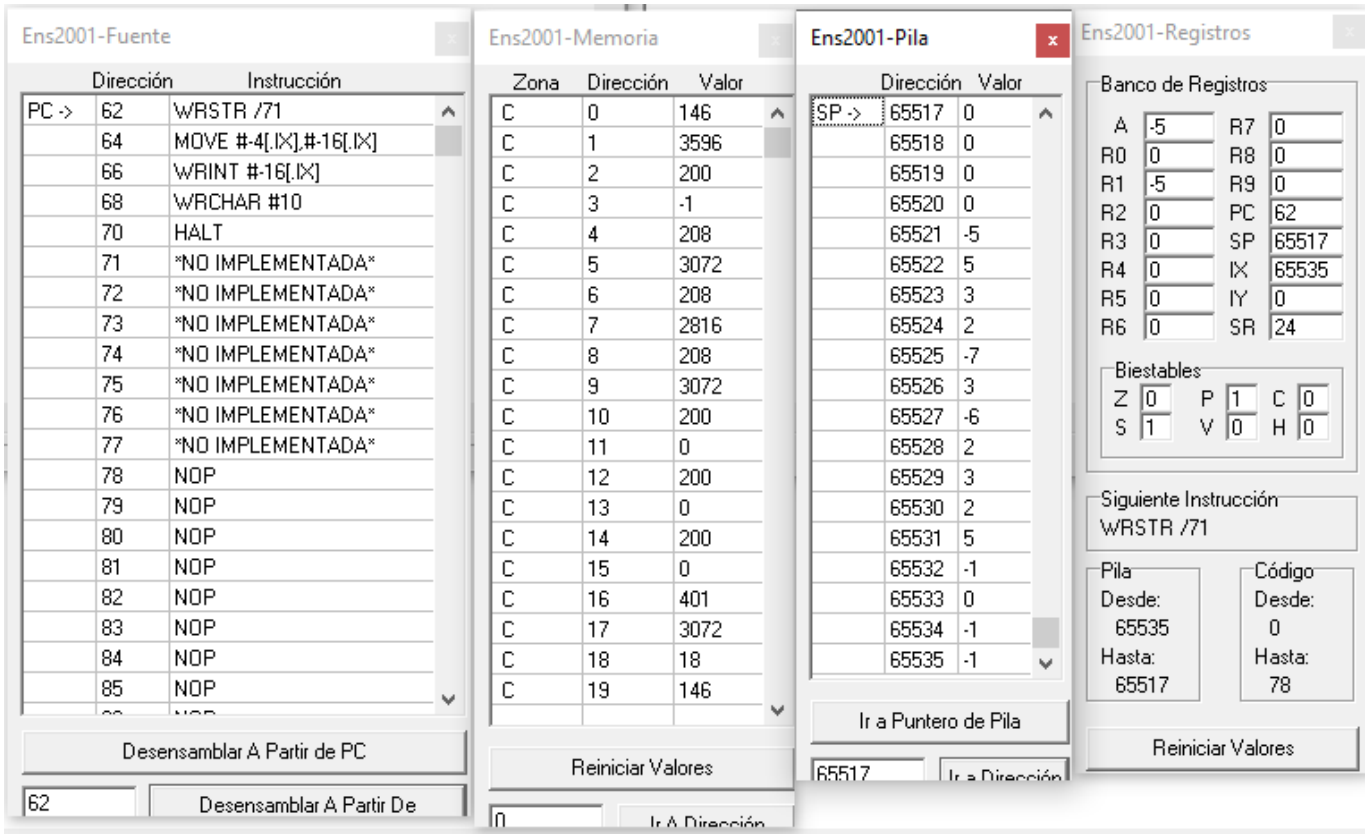
SUB .IX, #18
MOVE .A, .SP



Después de ejecutar las instrucciones que asigna valores a las variables “x” y “z”, y calcula el valor de la variable “suma”:

```
x = 2;
z = 3;
suma := x + z;
```

```
MOVE #2, #-7[.IX]
SUB .IX, #5
MOVE .A, #-8[.IX]
MOVE #-8[.IX], .R1
MOVE #-7[.IX], [.R1]
MOVE #3, #-9[.IX]
SUB .IX, #6
MOVE .A, #-10[.IX]
MOVE #-10[.IX], .R1
MOVE #-9[.IX], [.R1]
MOVE #-5[.IX], #-11[.IX]
MOVE #-6[.IX], #-12[.IX]
ADD #-11[.IX], #-12[.IX]
MOVE .A , #-13[.IX]
SUB .IX, #4
MOVE .A, #-14[.IX]
MOVE #-14[.IX], .R1
MOVE #-13[.IX], [.R1]
```



Una vez finalizada la ejecución del programa principal

```
WRSTR /L_0
MOVE /65535, /65523
WRINT /65523
WRCHAR #10
HALT
L_0 : DATA "SUMA = "
```