TABLA DE CONTENIDO

INTRODUCCION

¿Qué es mCOVI?

mCOVI (mini Computador Virtual) es una aplicación informática pensada para el aprendizaje de arquitectura y organización de computadores, permitiendo a los usuarios su manejo a través de una interfaz sencilla e intuitiva.

mCOVI simula el funcionamiento de varios procesadores RISC basados en la arquitectura SubDLX , que es un subconjunto de la arquitectura DLX propuesta por *Hennesy* y *Patterson* en sus libros [HePa93,96].

Estructura y uso de este documento

Este documento esta estructurado en los siguientes capítulos:

1. SubDLX : lenguaje maquina y ensamblador
2. Organizaciones SubDLX
3. Instalación y resolución de problemas
4. Primer contacto con mCOVI
5. Tutorial
6. Manual de referencia

Los dos primeros capítulos describen el fundamento teórico en el que se basa mCOVI, mientras que el resto de capítulos describen la utilización del programa mCOVI.

En la parte “teórica” se realiza una clara separación entre la arquitectura y la organización del computador, describiendo en el primer capítulo todo lo necesario para escribir programas correctos en el lenguaje ensamblador SubDLX, mientras que el segundo detalla y explica los componentes de las organizaciones que soporta mCOVI.

Para un buen manejo del programa, se recomienda la lectura en el orden establecido por este documento. Una vez instalado el programa (capítulo 3), el capítulo “Primer contacto con mCOVI” permite obtener una visión general del manejo de mCOVI, ampliándose un poco mas con el “Tutorial”, en el que se pretende describir las funcionalidades mas importantes de mCOVI. Por último, en el capítulo “Manual de referencia” se incluye una descripción detallada de todas las posibilidades en el manejo de mCOVI.

[HePa93] *ARQUITECTURA DE COMPUTADORES. Un enfoque cuantitativo*. J.L. Hennesy y D.A. Patterson, McGraw Hill 1993.

[HePa96] *COMPUTER ARCHITECTURE. A quantitive approach*. (Second Edition) J.L. Hennesy y D.A. Patterson, Morgan Kaufmann Publishers Inc 1996.

1.- SubDLX : lenguaje máquina y ensamblador

1.1. INTRODUCCION

La arquitectura SubDLX es un subconjunto de la arquitectura DLX de 32 bits propuesta por *Hennesy* y *Patterson* en sus libros [HePa93,96]. Es una arquitectura RISC con un modelo de cálculo *load/store* (carga/descarga). Esto significa que es necesario cargar en primer lugar los operandos desde memoria hacia el banco de registros. una vez cargados los registros, las instrucciones aritmético-lógicas pueden calcular nuevos valores sobre el mismo banco de registros. Finalizado el cálculo, los resultados deben moverse desde los registros hacia la memoria.

Estas arquitecturas se llaman también de tipo *load/store* (carga/descarga) ya que el acceso a los operandos de memoria sólo puede hacerse con estas operaciones. Salvo Pentium de Intel, el resto de arquitecturas de propósito general actuales son de este tipo: Alpha de Digital, MIPS-V de Silicon Graphics, PA 2.0 de Hewlett Packard, SPARC v9 de SUN y PowerPC de IBM/Motorola/Apple.

En los siguientes apartados se describe con detalle la arquitectura SubDLX.

[HePa93] *ARQUITECTURA DE COMPUTADORES. Un enfoque cuantitativo*. J.L. Hennesy y D.A. Patterson, McGraw Hill 1993.

[HePa96] *COMPUTER ARCHITECTURE. A quantitive approach*. (Second Edition) J.L. Hennesy y D.A. Patterson, Morgan Kaufmann Publishers Inc 1996.

1.2. ARQUITECTURA SubDLX

Los objetos principales de la arquitectura, datos e instrucciones, son objetos alineados de 32 bits. Es decir, su dirección de inicio debe ser múltiplo de cuatro, su tamaño en bytes. No está permitido automodificar el código, por lo cual todo intento de escritura en el espacio ocupado por instrucciones es ilegal. Una violación de estas reglas implica la terminación prematura del programa.

1.2.1.- Almacenes de datos

La numeración de los bits de los almacenes de datos se realiza asignando el número 0 al bit de menos peso, derecha de la palabra. Al bit de mas peso le corresponde el número 31, izquierda de la palabra.

Los almacenes de datos accesible en esta arquitectura son : el banco de registros y la memoria principal.

El banco de registros (BR) está formado por 32 registros de 32 bits de propósito general (RPG), denominándose `r0,r1,...,r31` (el contenido de `r0` es siempre cero). En esta arquitectura ningún registro tiene una funcionalidad específica, con lo que están todos disponibles para almacenar datos, direcciones o condiciones.

La memoria principal es direccionable al byte en modo *Big Endian* con una dirección de 32 bits. Todas las referencias a memoria se realizan a través de cargas o descargas entre memoria y el BR.

1.2.2.- Tipos de operaciones

Existen tres tipos de instrucciones :

- Operaciones arimético-lógicas (de la ALU)
- Operaciones de transferencia de datos
- Operaciones de control

Las **instrucciones de la ALU** son instrucciones registro-registro, es decir se realiza la operación entre valores dos provenientes del BR y se almacena el resultado también en BR

Se han considerado cuatro instrucciones de ALU : *suma* (`ADD`), *resta* (`SUB`),cálculo de condición *menor* (`SLT`) y cálculo de condición *menor sin signo* (`SLTU`).

Los cálculos de condición realizan una comparación entre el contenido de dos registros. Si se cumple dicha condición, se cara en el registro destino el valor “1”. Si la condición es falsa, se carga el valor cero.

Las **instrucciones de transferencia** de datos se realizan entre el BR y la memoria. Cualquiera de los registros puede ser cargado (`LW`) con un valor procedente de memoria, o su valor puede ser almacenado en memoria (`SW`). En el apartado siguiente se describe el modo de direccionamiento a utilizar para referenciar a la posición de memoria.

Las **instrucciones de control** permiten variar la secuencia de ejecución de instrucciones. Existen dos instrucciones de salto condicional y una que permite terminar la ejecución (`TRAP`).

Las instrucciones de salto condicional utilizan un registro de condición, por tanto no existen códigos de condición (*flags*). Se han considerado dos condiciones: *igual a cero* (`BEQZ`) y *distinto de cero* (`BNEZ`). Mas adelante se describe el modo de especificar la dirección de destino del salto.

No se han considerado las instrucciones de salto incondicional, debido a que éste tipo de saltos puede realizarse fácilmente con las existentes y la utilización de `r0` cuyo valor es siempre cero.

1.2.3.- Modos de direccionamiento

MODO REGISTRO

Para acceder a un registro de BR es necesario especificar el número del registro, de 0 a 32. Se utilizan 5 bits de la instrucción para direccionar un registro.

MODO Registro BASE + DESPLAZAMIENTO constante

Es el utilizado por las instrucciones de transferencia de datos (*load* y *store*). La dirección de memoria se obtiene sumando un registro (base) con el valor de 32 bits que resulta de la extensión de signo del entero de 16 bits (desplazamiento) que forma parte de la propia instrucción.

Si el registro base es `ro`, el direccionamiento se convierte en modo *absoluto*, mientras que si el desplazamiento es cero, se convierte en modo *indirecto*.

MODO RELATIVO AL PC

Se utiliza en las instrucciones de salto condicional. La dirección destino de salto se calcula sumando al PC incrementado la distancia que le separa del destino (positiva o no). Tal distancia se calcula multiplicando por cuatro la extensión de signo del campo de 16 bits que acompaña a la instrucción. En lenguaje ensamblador no aparece tal campo, sino la etiqueta de código destino de salto, siendo el programa ensamblador quien calcula la diferencia en número de instrucciones.

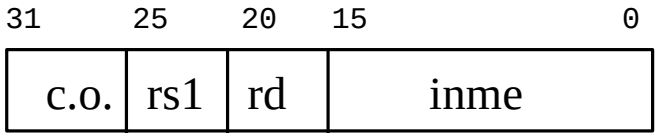
1.2.4.- Formatos de instrucciones

Todas las instrucciones se representan en lenguaje máquina con 32 bits, teniendo un código de operación principal de 6 bits. Existen tres formatos de instrucciones:

- Tipo I (inmediato)
- Tipo R (registro)
- Tipo J (*jump*)

TIPO I (inmediato)

La distribución de bits es la siguiente :

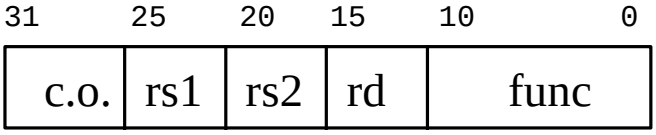


co : codigo de operación (6 bits)
rs1: registro fuente (5 bits)
rd : registro destino (5 bits)
inme : valor inmediato (16 bits)

Este formato de instrucciones lo utilizan las *load/store* y las instrucciones de control.

TIPO R (registro)

La distribución de bits es la siguiente :



co : codigo de operación (6 bits)
rs1: registro fuente 1(5 bits)
rs2: registro fuente 2(5 bits)
rd : registro destino (5 bits)
func: codificación operación (11 bits)

Este formato de instrucciones lo utilizan las instrucciones aritmético lógicas. Todas siguen el mismo esquema:

rd ← rs1 *func* rs2

De los 11 bits del campo func, solo son significativos para SubDLX los 6 de menor peso.

TIPO J (*jump*)

La distribución de bits es la siguiente :

Este formato de instrucciones solo lo utiliza la instrucción que permite la finalización del programa, y no tiene ninguna relevancia el campo desplazamiento, que deberá tener el valor cero.

1.3.
LENGUAJE ENSAMBLADOR SubDLX

En este apartado se dan las reglas sintácticas del lenguaje ensamblador soportado por mCOVI. En el caso que se intente ensamblar un progama que no sea correcto, se informa al usuario de los errores cometidos.

Un progama en lenguaje ensamblador es una secuencia de instrucciones y directivas. En los apartados siguientes se describe su sintaxis.

1.3.1.- Formato de instrucciones en ensamblador SubDLX

Una instrucción consta de 5 campos: etiqueta, operación, operandos, comentario y terminador. Solo el terminador es necesario. La figura siguiente muestra la formato de una instrucción:

```
[etiqueta] [operación]   [operandos]   [comentario]    terminador
inibucle:      add       r1, r2, r3    ; incrementa r1  <RETURN>
```

El programa ensamblador impone las siguientes restricciones:

- No existe carácter de continuación de línea
- Los campos de una instrucción deben estar separados por espacios o tabuladores
- Todos los campos de una instrucción deben aparecer en la misma línea

El campo **etiqueta** permite referenciar la dirección de la instrucción dentro del programa. Dos instrucciones no pueden tener la misma etiqueta. El fragmento de programa siguiente muestra el uso típico de las etiquetas para a la realización de saltos condicionales.

```
inibucle:   slt       r1, r2, r3
          .........
            bnez      r4,inibucle
```

El segundo campo de la instrucción es el campo **operación**, que contiene un mnemotécnico correspondiente a la operación a realizar. Las operaciones son de dos tipos : mnemotécnicos de instrucción y directivas ensamblador. Los primeros son nombre simbólicos correspondientes a instrucciones de lenguaje máquina, mientras que las segundas son comandos que se dan al ensamblador para la reserva e inicialización de almacenamiento.

El campo **operando** consta de uno o mas operandos separados por comas. El número y tipo de los operandos depende de la operación a realizar. Normalmente cada operando especifica un registro de la máquina o un modo de direccionamiento.

El campo **comentario** es opcional, comienza con ‘;’ y termina con <return>.

1.3.2.- Directivas en ensamblador SubDLX

Las directivas indican al programa ensamblador que realice reservas o inicializaciones. Este apartado describe las siguientes directivas:

- Directivas de división de zonas de memoria
- Directivas de almacenamiento
- Directivas de alineamiento
- Directivas de nombres simbólicos.

El espacio de direcciones en los programas esta dividido en dos zonas: la de datos y la código. Existen dos directivas que sirven para marcar su inicio: `.data` y `.text`. La dirección de comienzo de cada zona se puede especificar con un argumento:

```
.data [direccion]
.text [direccion]
```

Las instrucciones de acceso a memoria solo pueden referenciar a direcciones situadas en la zona de datos. Si durante la ejecución del programa ensamblador se pretende acceder a la zona de código, se genera un error de direccionamiento (ver capítulo 3).

Si se desea reservar espacio en memoria hay que utilizar la directiva .**space**. Dicha directiva reserva una serie de bytes en la zona de datos. El número de bytes es el argumento de la directiva:

```
.space nbytes
```

Existen directivas que permiten almacenar una serie de valores en la zona de datos. Cada directiva almacena un tipo de valor diferente : cadenas de caracteres, terminadas con el byte 0 (`.asciiz`) o no (`.ascii`) y palabras de 32 bits (`.word`). Los valores a almacenar se listan en los argumentos:

```
.ascii  "cadena"  [,"cadena"[,"cadena"]....]
.asciiz "cadena"  [,"cadena"[,"cadena"]....]
.word   expresion [,expresion[,expresion]....]
.half   expresion [,expresion[,expresion]....]
.byte   expresion [,expresion[,expresion]....]
```

Con la directiva `.align` se consigue que el siguiente de la zona de datos se alinee. El limite viene especificado en función de su argumento. Su sintaxis es la siguiente:

```
.align n
```

lo que provoca que el siguiente dato se alinea sobre un limite de 2^n bytes.

1.3.3.- Expresiones en ensamblador SubDLX

Las sintaxis de la expresiones es como en el lenguaje C. Los números se pueden especificar en notación decimal, o notación hexadecimal si los dos primeros caracteres del número son ‘0x’.

La forma mas simple de una expresión es un número. Ahora bien, generalizando, una expresión de dirección puede estar compuesta de:

- Números
- Símbolos (definidos en el programa)
- Operadores : *, /, +, -, <<, >>, &, |, %, ^, ~ (con la misma precedencia que en C)
- Paréntesis para agruparlos

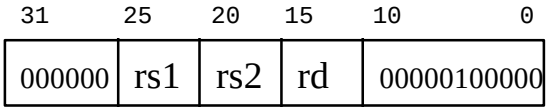
1.4.
REPERTORIO DE INSTRUCCIONES SubDLX

1.4.1. Instrucciones aritméticas

ADD

Lenguaje ensamblador: `ADD rd, rs1, rs2`

Lenguaje máquina (tipo R) :



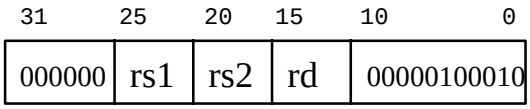
Operación: rd = rs1 + rs2

Descripción: Suma el contenido de dos registros de 32 bits dejando el resultado en otro registro.

SUB

Lenguaje ensamblador: SUB `rd, rs1, rs2`

Lenguaje máquina (tipo R) :



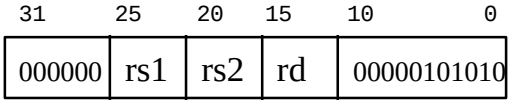
Operación: rd = rs1 - rs2

Descripción: Resta el contenido de dos registros de 32 bits dejando el resultado en otro registro.

SLT

Lenguaje ensamblador: SLT `rd, rs1, rs2`

Lenguaje máquina (tipo R) :



Operación:

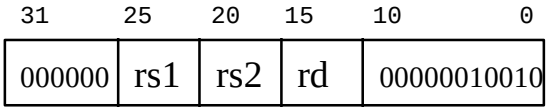
Si rs1 < rs2 (con signo)
entonces rd = 1
sino rd = 0;

Descripción: Inicializa con valor 1 el registro destino si el valor del primer registro es menor (con signo) que el valor del segundo registro. **rs1** y **rs2** se consideran números enteros.

SLTU

Lenguaje ensamblador: SLTU `rd, rs1, rs2`

Lenguaje máquina (tipo R) :



Operación:

Si rs1 < rs2 (sin signo)
entonces rd = 1
sino rd = 0;

Descripción: Inicializa con valor 1 el registro destino si el valor del primer registro es menor (sin signo) que el valor del segundo registro. **rs1** y **rs2** se consideran números naturales.

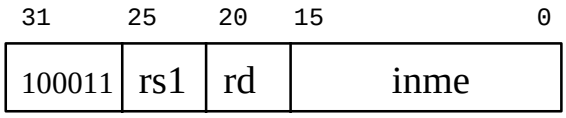
.

1.4.2. Instrucciones de transferencia de datos

LW

Lenguaje ensamblador: `LW rd, inme(rs1)`

Lenguaje máquina (tipo I) :



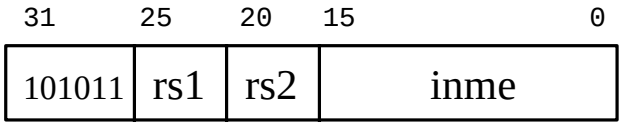
Operación: rd = memoria [sex(inme)+rs1]

Descripción: Carga una palabra de memoria en un registro del banco de registros. Utiliza direccionamiento base+desplazamiento para calcular dicha dirección. **sex** es el operador de extensión de signo de 16 a 32 bits.

SW

Lenguaje ensamblador: `SW inme(rs1),rs2`

Lenguaje máquina (tipo I) :



Operación: memoria [sex(inme)+rs1] = rs2

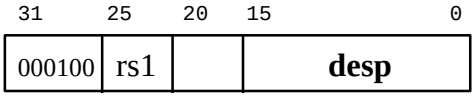
Descripción: Almacena una palabra en memoria con un valor contenido en un registro. Utiliza direccionamiento base+desplazamiento para calcular dicha dirección. **sex** es el operador de extensión de signo de 16 a 32 bits.

1.4.3. Instrucciones de salto condicional

BEQZ

Lenguaje ensamblador: BEQZ `rs1, <etiqueta>`

Lenguaje máquina (tipo I) :



`desp` ese un desplazamiento en número de instrucciones sobre el valor del PC de la propia instrucción (PCinst). Por ejemplo, si una instrucción de salto condicional quisiera saltar a si misma, el valor sería 0.

Operación:

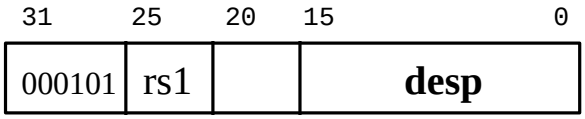
Si rs1 = 0
entonces PCdst = PCinst + 4*sex(desp)

Descripción: Se salta a la instrucción referenciada por <etiqueta> en el caso que el contenido del registro sea cero.

BNEZ

Lenguaje ensamblador: BNEZ `rs1, <etiqueta>`

Lenguaje máquina (tipo I) :



`desp` supone un desplazamiento en número de instrucciones sobre el valor del PC de la propia instrucción (PCinst). Por ejemplo, si una instrucción de salto condicional quisiera saltar a si misma, el valor sería 0.

Operación:

Si rs1 not = 0
entonces PCdst = PCinst + 4*sex(desp

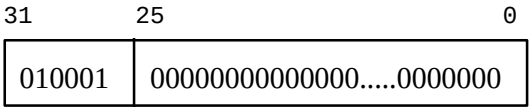
Descripción: Se salta a la instrucción referenciada por <etiqueta> en el caso que el contenido del registro sea distinto de cero.

1.4.4. Instrucciones de final de programa

TRAP

Lenguaje ensamblador: TRAP 0

Lenguaje máquina (tipo J) :



Operación: Termina la ejecución del programa

1.5. EJEMPLO

```
;*********************************************************************
;***********       miniCOVI          *******************************
;*********************************************************************
;
;*********** Programa que realiza el producto de dos numeros  ******
;*********** naturales, mediante sumas sucesivas               *****
;*********************************************************************
;
;**** Los factores se encuentran en memoria y el resultado    ******
;**** tambien se almacena en memoria                          ******
;*********************************************************************

         .data

Num1   : .word           5
Num2  :  .word           10
Resul:   .space          4

        .text
        .global main

main:   lw  r3, Num1      ; Cargo valores en registros
        lw  r4, Num2

Loop:
        slt r1, r2, r3    ; Control de final del bucle
        beqz r1 , FINAL
        add r2, r2, r1
        add r5, r5, r4   ; Suma sucesiva
        bnez r1, LOOP

Final:  sw Resul, r5     ; Almaceno el resultado
        trap 0
```

2.- ORGANIZACIONES SubDLX

2.1. INTRODUCCION : ¿Qué se entiende por organización?

En este documento, se entiende por “Organización” del procesador a la descripción de las unidades funcionales que lo componen y la manera en que se interconectan para soportar una arquitectura. El comportamiento de la Unidad de Control (UC) en función de sus entradas (generación de secuencias de control), también forma parte de la Organización del procesador.

En nuestro caso, la arquitectura SubDLX puede ser soportada por cuatro organizaciones diferentes. A continuación se detallan los elementos comunes a todas las organizaciones, para luego comentar los específicos.

2.2. TIPOS DE ORGANIZACION SubDLX

Todo procesador consta de una ruta de datos y de una UC. Hemos considerado dos rutas de datos, una con un tiempo de ciclo elevado (*ciclo largo*) y otra con un tiempo de ciclo reducido (*ciclo corto*). También hemos considerado dos opciones para diseñar la UC: *cableada* y *microprogramada*.

Por tanto, mCOVI soporta cuatro procesadores diferentes de la arquitectura subDLX:

- Ruta de Datos de ciclo largo con control cableado (**CL.CAB**)
- Ruta de Datos de ciclo largo con control µprogramado (**CL.µ)**
- Ruta de Datos de ciclo corto con control cableado (**CC.CAB**)
- Ruta de Datos de ciclo corto con control µprogramado (**CC.µ)**

Para ejecutar un programa, el usuario debe seleccionar una de las organizaciones anteriores. En el control microprogramado (CL.µ y CC.µ) puede ampliarse el microcódigo, es decir la UC no solo es microprogramada, sino también microprogramable.

2.3. ELEMENTOS DE LAS RUTAS DE DATOS

En este apartado se van a describir los elementos de la ruta de datos comunes a todas las organizaciones. Para cada elemento se detallan su comportamiento, así como sus señales de entrada, control y salida.

2.3.1.- Memoria Principal

La memoria principal está constituida por un único espacio de direcciones de 32 bits, en el que residen datos e instrucciones. Ahora bien, la ruta de datos accede a instrucciones y datos que están en memorias separadas (organización tipo Harvard). Puesto que está prohibido el código automodificable, durante la ejecución de un programa, la memoria de instrucciones es de sólo lectura.

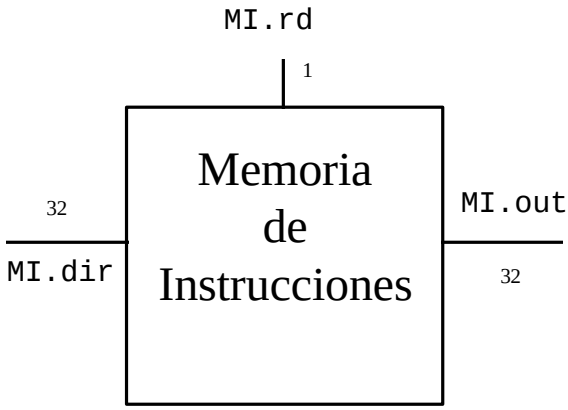
El motivo de esta separación es aumentar la claridad de la ruta de datos, y facilitar el tránsito a los procesadores segmentados, que acceden en paralelo a memorias cache separadas.

MEMORIA DE INSTRUCCIONES

La memoria de instrucciones (**MI)**, tiene una señal de *permiso de lectura* (`MI.rd`) mediante la cual la UC indica si hay que leer en el ciclo actual.

La *dirección* a leer llega mediante la entrada de 32 b `MI.dir`.

En el ciclo en que esta activa `MI.rd`, la memoria de instrucciones coloca en la *salida* `MI.out` la palabra referenciada. Por tanto la operación de lectura tiene una latencia de un ciclo.

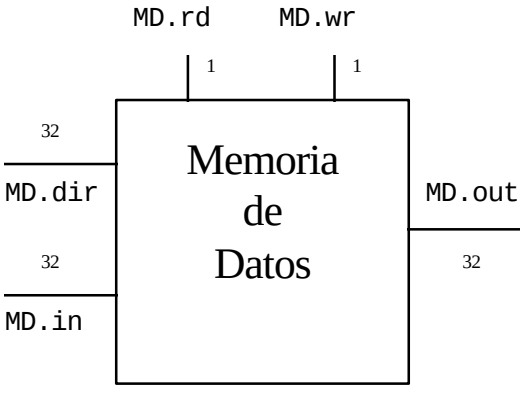


MEMORIA DE DATOS

La memoria de datos (**MD)**, tiene una señal de *permiso de lectura* (`MD.rd`) y una señal de *permiso de escritura* (`MD.wr`).

Ambas operaciones tienen una latencia de un ciclo.

La MD tiene dos señales de entrada, una de *dirección* (`MD.dir`) y la otra de *dato a escribir*(`MD.in`)

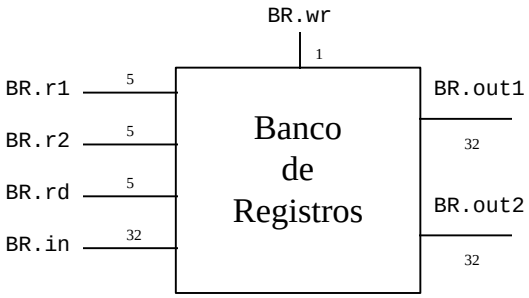


2.3.2.- Banco de Registros

El Banco de Registros (**BR**) está constituido por 32 registros de 32, pudiéndose realizar operaciones de lectura y escritura sobre ellos. Se denominan `r0,r1,...,r31`, y el contenido de `r0` es siempre cero. Al comienzo de cada ciclo, se inicia la lectura de dos registros cuyos números se codifican en las entradas de *dirección de lectura* `BR.r1` y `BR.r2`.

Los valores de los registros leídos, se colocan en las *salidas* `BR.out1` y `BR.out2`.

La escritura en un registro del BR está sincronizada por el final de ciclo (flanco ascendente), y se realiza si el permiso de escritura (`BR.wr`) está activo



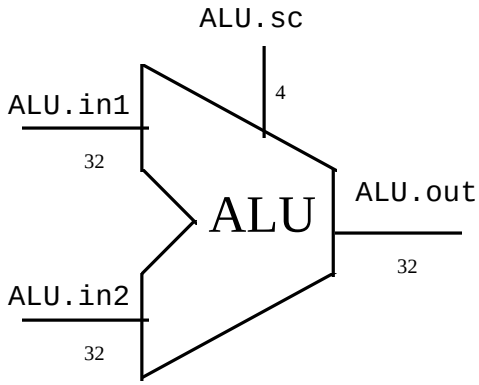
El número de registro a escribir está codificado la entrada de *dirección de escritura* (`BR.rd`), mientras que el dato a escribir está en la *entrada de escritura* `BR.in`[1] .

2.3.3.- Unidad Aritmético Lógica

La Unidad Aritmético Lógica, en adelante ALU, es la encargada de realizar las operaciones aritmético lógicas, con una latencia de un ciclo.

Los operandos sobre los que se realiza la operación están en las *entradas* `ALU.in1` y `ALU.in2`. El resultado aparece en la *salida* `ALU.out`.

La *operación* a realizar está gobernada por la entrada de control `ALU.sc`, que con sus 4 bits codifica 16 operaciones distintas. Esto parece excesivo, ya que en el repertorio SubDLX solo existen cuatro operaciones artimético-lógicas.



Sin embargo, esta ampliación permite al usuario implementar nuevas instrucciones en los procesadores microprogramables (`CC.μ` y `CL.μ`).

El origen de la señal `ALU.sc` es diferente según la organización: para las que tienen control μprogramado, su origen es la propia UC; mientras que con control cableado, la señal viene de un acondicionador conectado a IR.

[1] Formalmente: si en el ciclo [i] está activo el permiso de escritura y `BR.rd`[i] = j, entonces rj[i+1] = `BR.in`[i]

En la tabla siguiente se relacionan las diferentes operaciones que realiza la ALU, de las cuales las cuatro primeras son las que van asociadas al repertorio SubDLX.

ALU.sc	Operación	Descripción de la operación
0000	ADD	Suma
0001	SUB	Resta
0010	SLT	Cálculo condición *menor*
0011	SLTU	Cálculo condición *menor sin signo*
0100	AND	Producto lógico
0101	OR	Suma lógica
0110	XOR	Suma exclusiva lógica
0111	SLL	Desplazamiento lógico a la izquierda
1000	SRA	Desplazamiento aritmético a la derecha
1001	SRL	Desplazamiento lógico a la derecha
1010	SEQ	Cálculo condición *igual*
1011	SNE	Cálculo condición *distinto*
1100	SLE	Cálculo condición *menor o igual*
1101	SLEU	Cálculo condición *menor o igual sin signo*
1110		
1111		

La descripción de las operaciones Cálculo *`condición`*, es la siguiente:

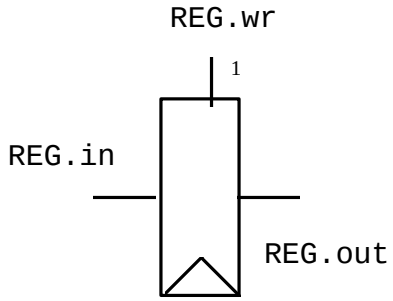
```
Si ALU.in1 condición ALU.in2
    entonces ALU.out = 1
    sino     ALU.out = 0;
```

En las operaciones de desplazamiento, el número de bits a desplazar está en `ALU.in2`, mientras que el valor a desplazar está en `ALU.in1`. En las operaciones de desplazamiento lógico se rellenan las posiciones libres con ceros, mientras que en el desplazamiento aritmético a la derecha se replica el bit mas significativo en las posiciones libres.

2.3.4- Registros

Todos los registros de la ruta de datos son de 32 bits y su funcionamiento sigue las mismas pautas.). El *contenido* del registro se coloca en la señal de *salida* (`REG.out`).

La escritura está sincronizada por final de ciclo (flanco ascendente). Se necesita *permiso de escritura* (señal de control `REG.wr`), para almacenar el valor de *entrada* (`REG.in`)[2].



IR : Registro de Instrucción

El IR (*Instruction Register*) contiene la instrucción a ejecutar. Los 32 bits de salida de IR se agrupan en campos que van a parar a varios componentes de la ruta de datos o de la UC.

En la tabla adjunta se detallan los campos de salida del IR.

Señal de salida	Numero de bits	Bits de IR
IR.CO	6	31..26
IR.CO[0]	1	26
IR.rs1	5	25..21
IR.rs2	5	20..16
IR.rd	5	15..11
IR.inme	16	15..0
IR.func[5..0]	6	5..0

PC : Contador de Programa

El PC (*Program Counter*) contiene la dirección de memoria de la siguiente instrucción a ejecutar. Este registro se actualiza en el último ciclo de cada instrucción

DMAR: Registro de direcciones de la Memoria de Datos

El DMAR (*Data Memory Address Register*) contiene la dirección de la memoria de datos sobre la que se va a realizar un acceso, ya sea lectura o escritura.

MDRin: Registro de entrada a la Memoria de Datos

El MDRin (*Memory Data Register*) contiene el dato que se va a escribir en la memoria de datos.

[2] Formalmente: si `REG.wr`[i], entonces REG.out[i+1] = `REG.in`[i]

2.3.5- Evaluador de cero

El Evaluador de cero se utiliza exclusivamente en las instrucciones de salto condicional (`BEQZ` y `BNEZ`), para controlar si se toma el salto o no, por lo que su salida está conectada a la UC.

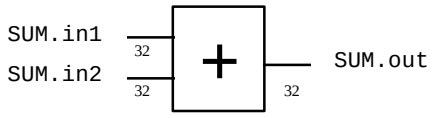
Tiene una señal de entrada de 32 bits (`EVAL.in`), y una señal de salida de 1 bit (`EVAL.out`). Su comportamiento obedece al siguiente pseudocódigo:

```
Si EVAL.in = 0
    entonces EVAL.out = 1
    sino     EVAL.out = 0;
```

2.3.6- Sumadores

Los sumadores que existen en la ruta de datos realizan la suma de dos operandos de entrada (SUM_.in1 y SUM_.in2), poniendo el resultado en la salida (SUM_.out).

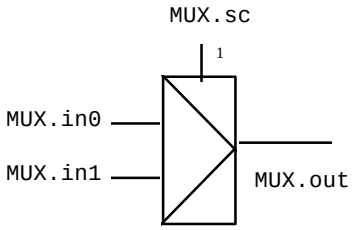
La latencia es de un ciclo y no tienen conexión con la UC, por lo que realizan su función en todos los ciclos.



2.3.7- Multiplexores

Todos ellos tienen dos canales de entrada (`MUX_in0` y `MUX_in1`), un canal de salida (`MUX_.out`) y una señal de control (`MUX_.sc`).

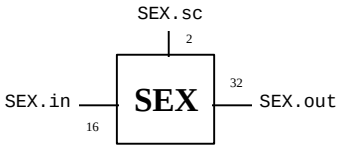
La anchura de los canales varía en función de la misión del multiplexor, teniendo todos 32 bits excepto el que alimenta a `BR.rd` que es de 5 bits.



2.3.8- SEX: Extensor de signo

El SEX (*Sign Extensor*) es un bloque combinacional que extiende el signo de su entrada de 16 bits (`SEX.in`) en su salida de 32 bits (`SEX.out`) con una latencia de un ciclo.

El SEX de `xx.µ` posee además una señal de control (`SEX.sc`) de dos bits que indica el tipo de extensión a realizar.



Gracia a esta complejidad añadida, pueden microprogramarse instrucciones pertenecientes al repertorio DLX (`ANDI, ORI y XORI`).

Existen 3 tipos de extensiones, según aparece en la siguiente tabla :

SEX.sc	Tipo de extensión
00	Inserta ceros
01	Extensión de signo
10	
11	Inserta unos

2.3.9- Multiplicador por 4

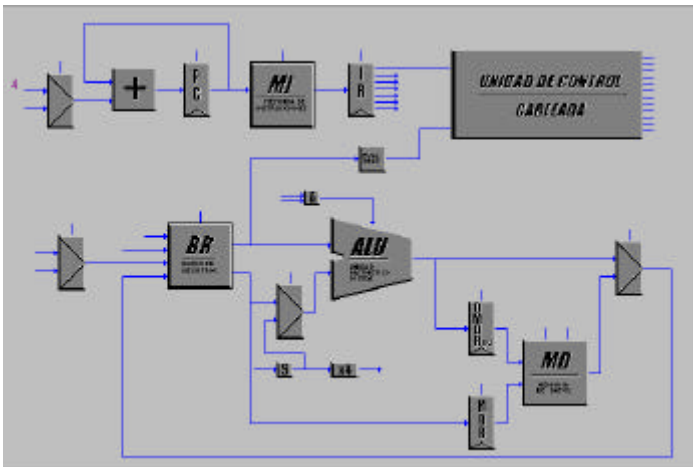
Este elemento solo posee una señal de entrada y una de salida, ambas de 32 bits. Multiplica por cuatro mediante una simple conexión de la salida con la entrada desplazada dos bits a la izquierda. En los dos bits de menos peso de la salida se colocan ceros.

2.4.
RUTA DE DATOS DE CICLO LARGO

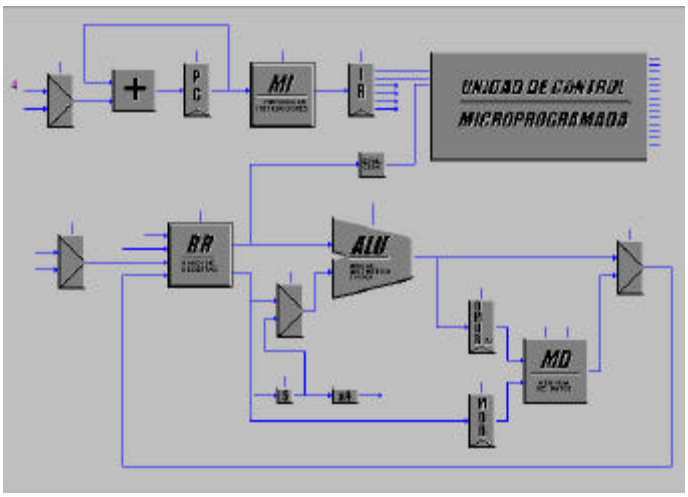
Este tipo de ruta de datos necesita un tiempo de ciclo relativamente grande, pero también reduce el número de ciclos por instrucción.

Las organizaciones CL.CAB y CL.µ son idénticas salvo en el control de SEX y de ALU. En CL.CAB no existe señal de control en SEX y en CL.µ si. En cuanto al control de ALU, en CL.µ la señal ALU.sc tiene el origen en la UC, mientras que en CL.CAB aparece un acondicionador en la ruta de datos que genera dicha señal.

La representación gráfica de CL.CAB es la siguiente:



La representación gráfica de CL.µ es la siguiente:



2.5. RUTA DE DATOS DE CICLO CORTO

Este tipo de ruta de datos admite un tiempo de ciclo mas pequeño, pero incrementa el numero de ciclos por instrucción.

La reducción del tiempo de ciclo se consiguen al insertar una serie de registros en la ruta de datos que provocan la disminución de actividad en cada ciclo. Se han añadido cuatro registros: `RA, RB, RT y MDRout`[3].

El registro `RA` almacena la salida 1 del Banco de Registros, antes de servir como primer operando en la operación de ALU. El registro `RT` se carga con el resultado de ALU antes de escribirse en el BR. El Registro `MDRout` se carga con el dato leído de la memoria de datos antes de escribirse también en el BR.

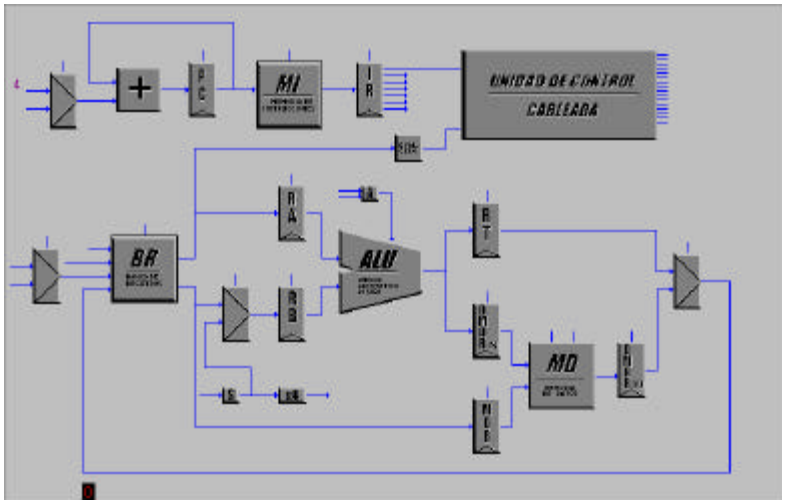
El registro `RB` tiene situación diferente según el tipo de control. En `CC.CAB`, la salida de RB se conecta directamente con la entrada 2 de la ALU, y el valor que se almacena procede del multiplexor. Mientras que en la organización CC.µ, el valor almacenado en RB es la salida 2 del Banco de registros y la salida de RB se conecta con una entrada del multiplexor, siendo la salida de éste la que se conecta con la entrada 2 de la ALU

Esta situación diferente de RB se realiza para minimizar el numero de ciclos por instrucción en la organización `CC.`

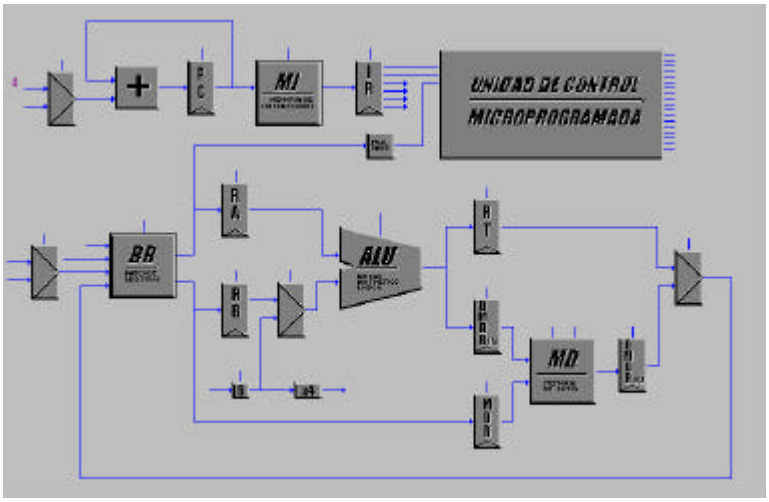
Las diferencias de control en ALU y SEX son las ya descritas para la ruta de datos de ciclo largo.

[3] Aunque no es objetivo de mCOVI, esta ruta de datos admite un funcionamiento segmentado sin mas que cambiar la UC..

La representación gráfica de `CC.CAB` es la siguiente:



La representación gráfica de `CC.μ` es la siguiente:



2.6. UNIDAD DE CONTROL CABLEADA (UCC)

La Unidad de Control Cableada es un circuito secuencial : sus entradas se transforman en una secuencia de salidas que son las señales de control de la ruta de datos.

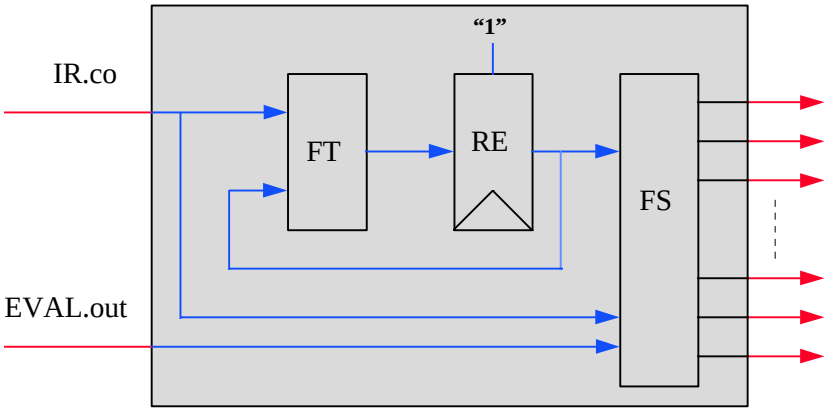
2.6.1.- Descripción general y componentes

La UCC es un autómata de estados finitos (o máquina) de tipo *Mealy*. Está compuesta por los siguientes elementos:

- Función de transición (**FT**)
- Registro de estado (**RE**)
- Función de salida (**FS**)

Las entradas a la UCC son `IR.co` y `EVAL.out`, mientras que las salidas son todos los puntos de control de la ruta de datos.

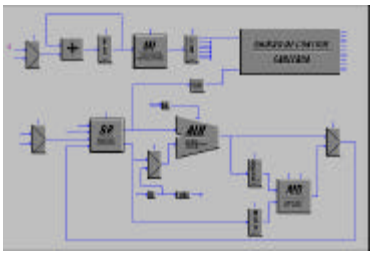
Su organización es la siguiente:



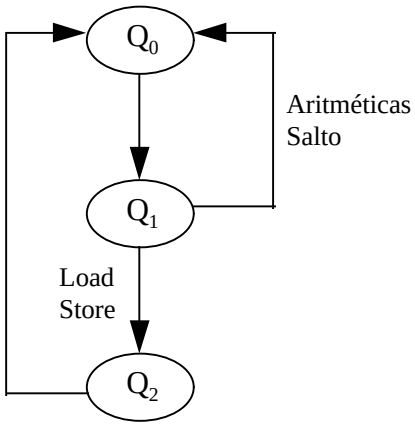
La **FT** genera el estado futuro a partir del estado actual y de la entrada correspondiente al código de operación. El **RE** almacena el estado de la U.C. La **FS** genera las señales de control a partir del estado actual, del código de operación y de la salida del evaluador de cero.

Las características detalladas de estos componentes en las dos rutas de datos tratadas se abordan en los apartados posteriores.

2.6.2.- UCC para ruta de Datos de ciclo largo



El diagrama de transición entre estados es el siguiente:



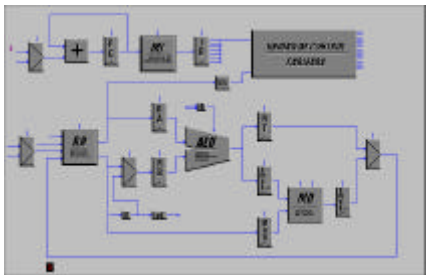
El RE codifica el estado con un valor de 2 bits.

En la tabla siguiente se muestra el comportamiento de salida de la UCC.

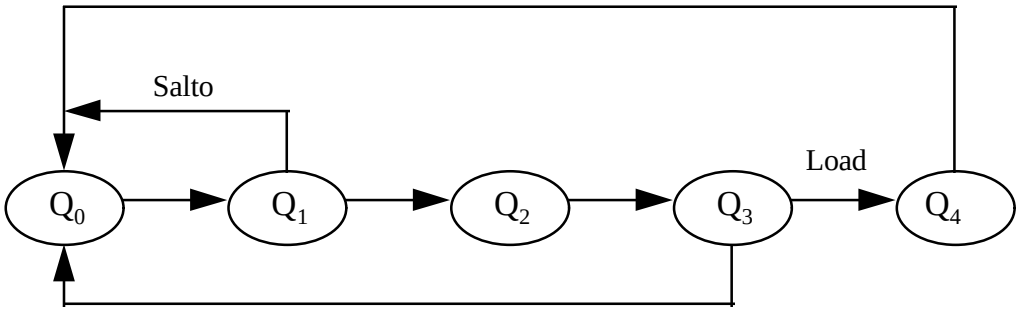
	Q0	Q1					Q2	
		Aritméticas	Load	Store	Salto cumplido	Salto NO cumplido	Load	Store
MI.rd	1							
MD.rd							1	
MD.wr								1
PC.wr		1			1	1	1	1
IR.wr	1							
BR.wr		1					1	
DMAR.wr			1	1				
MDRin_wr				1				
MUXPC.sc		0			1	0	0	0
MUXBRrd.sc		0					1	
MUXBRin.sc		0					1	
MUXALU.sc		0	1	1				

En el caso de las señales de control de los multiplexores, los valores que no aparecen en la tabla son cero aunque no influyan para nada. En el resto de señales los valores que no aparecen son cero, que significa la inexistencia de permiso para realizar la operación.

2.6.3.- UCC para ruta de Datos de ciclo corto



El diagrama de transición entre estados es el siguiente:



Aritméticas - Store

En este diagrama se observa que existen 5 posibles estados, por lo que se necesitan 3 bits para representar el estado.

En la tabla siguiente se muestra el comportamiento de salida de la UCC.

	Q_0	Q_1					Q_2			Q3			Q4
		Aritméticas	Load	Store	Salto cumplido	Salto NO cumplido	Artiméticas	Load	Store	Artiméticas	Load	Store	
MI.rd	1												
MD.rd													
MD.wr												1	
PC.wr					1	1				1		1	1
IR.wr	1												
BR.wr										1			1
DMAR.wr								1	1				
MDRin.wr									1				
MDRout.wr											1		
RA.wr		1	1	1									
RB.wr		1	1	1									
RT.wr							1						
MUXPC.sc					1	0				0		0	0
MUXBRrd.sc										0			1
MUXBRin.sc										0			1
MUXALU.sc		0	1	1									

En el caso de lase señales de control de los multiplexores, los valores que no aparecen en la tabla son cero aunque no influyan para nada. En el resto de señales los valores que no aparecen son cero, que significa la inexistencia de permiso para realizar la operación.

2.7.
UNIDAD de CONTROL µProgramada/µPrograMable (UCµ)

La microprogramación fué ideada por el profesor M.V. Wilkes en la Universidad de Cambridge sobre 1950, para simplificar el diseño, depuración y modificación de la UC de los procesadores CISC que se adivinaban en el futuro. La microprogramación se basa en especificar el comportamiento de una instrucción máquina mediante una secuencia de *microinstrucciones* (µinstrucciones).

2.7.1.- Descripción general y componentes

Cada µinstrucción se ejecuta en un ciclo de reloj, con lo que la temporización no sufre grandes modificaciones con respecto a la UCC. Una µinstruccion está compuesta por los siguientes campos:

- varias señales de control de 1 bit (diferentes en número según ciclo corto o largo)
- control de la ALU (4 bits)
- control de SEX (2 bits)
- secuenciamiento de µinstrucciones (2 bits).

Para cada instrucción en ensamblador se ejecutan una serie de µinstrucciones. Las dos primeras µinstrucciones (0 y 1) son comunes a todas ellas y consisten en las fases de búsqueda y decodificación de la instrucción, respectivamente.

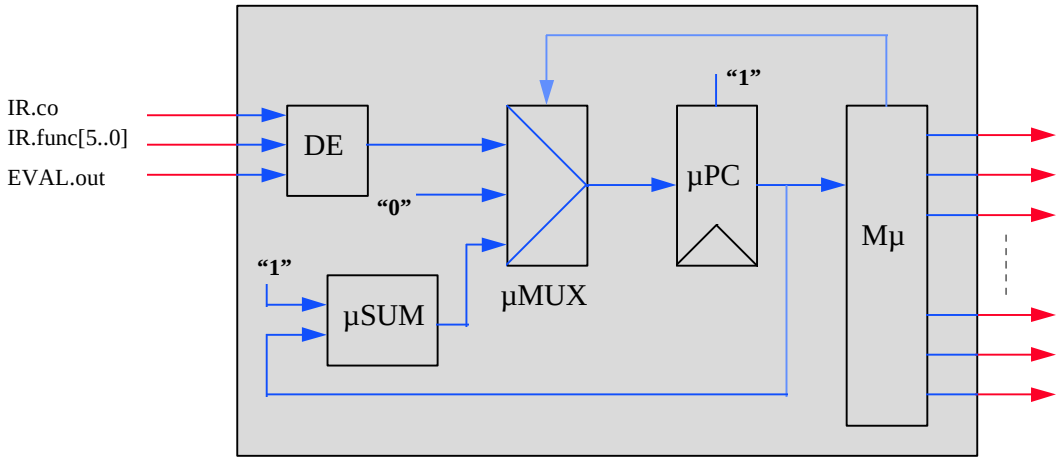
Al decodificar una instrucción máquina se obtiene el número de la siguiente µinstrucción a ejecutar. Esta selección se realiza en base a las entradas `IR.co` y `IR.func[5..0]`.

La UCµ esta compuesta de los siguientes elementos:

- Decodificador-Evaluador (**DE**)
- Memoria de µcódigo (**Mµ**)
- µContador de programa (**µPC**)
- µMultiplexor (**µMUX**)
- µSumador (**µSUM)**
- Función de salida (**FS**)

El último campo de la µinstrucción (secuenciamiento) se utiliza para controlar el multiplexor de entrada a µPC y seleccionar la µinstrucción siguiente. La entrada `EVAL.out` se considera para determinar la reacción en caso de instrucciones máquina de salto condicional.

La estructura interna de la UCμ es la siguiente:



Las μinstrucciones están almacenadas en la **Mμ,** la cual tiene una capacidad máxima de 64 μinstrucciones, con lo que el **μPC** tiene 6 bits.

De las 64 posiciones de Mμ existen unas que tiene valores fijos (almacenadas en ROM) y que constituyen el intérprete del repertorio SubDLX. El resto (almacenadas en RAM) son cargables por el usuario, con lo que la UCμ es también μprogramable.

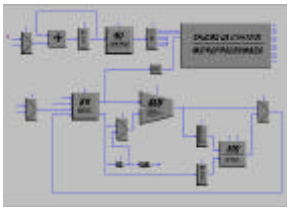
El **μMUX** selecciona el número de la μinstrucción siguiente, en base al campo secuenciamiento de cada μinstrucción. El μMUX tiene tres entradas: la primera proveniente del DE, la segunda fija con valor cero, y la tercera viene de μSUM.

El **DE** produce el número de la siguiente μinstrucción a ejecutar. Puede implementarse como una ROM o una PLA accedidas por la concatenación de (`IR.co, IR.func[5..0]` y `EVAL.out`). Pueden añadirse mas entradas para microprogramar nuevas instrucciones máquina (ver 2.7.4).

Cuando se ejecuta la última μinstrucción de cada instrucción, el μMUX selecciona el valor cero para iniciar la fase de búsqueda de la siguiente instrucción. Para ejecutar la siguiente μinstrucción en secuencia, se debe seleccionar la μdirección proveniente de μSUM.

Las características detalladas de estos componentes en las dos rutas de datos se abordan a continuación.

2.7.2.- UCμ para ruta de Datos de ciclo largo



El Decodificador-Evaluador de la UCμ para la ruta de datos de ciclo largo, posee los siguientes valores internos:

		Número de μ instrucción		
IR.co	IR.func[5..0]	Si EVAL.out = 0	Si EVAL.out = 1	Comentario
000000	100000	000010	000010	Linea para ADD (μinst = 2)
000000	100010	000011	000011	Linea para SUB (μinst =3)
000000	101010	000100	000100	Linea para SLT (μinst =4)
000000	010010	000101	000101	Linea para SLTU (μinst = 5)
100011		000110	000110	Linea para LW (μinst = 6)
101011		001000	001000	Linea para SW (μinst = 8)
000100		001010	001011	Linea para BEQZ (μinst = 10 si salto no tomado; y μinst=11 si salto tomado)
000101		001011	001010	Linea para BNEZ (μinst = 11 si salto tomado; y μinst=10 si salto no tomado)

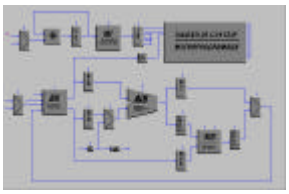
Las señales de control de 1 bit de **CL.μ** son las siguientes:

```
0.- MUXALU.sc      6.- DMAR.wr
1.- MUXBRrd.sc     7.- IR.wr
2.- MUXBRin.sc     8.- PC.wr
3.- MUXPC.sc       9.- MD.wr
4.- BR.wr          10.- MD.rd
5.- MDRin.wr       11.- MI.rd
```

El contenido de la Mμ para la ruta de datos de ciclo largo es la siguiente :

	Señales de control de 1 bit															
μInst.	11	10	9	8	7	6	5	4	3	2	1	0	ALU.sc	SEX.sc	μMUX.sc	Comentario
0	1	0	0	0	1	0	0	0	0	0	0	0	0000	01	10	Fase de búsqueda
1	0	0	0	0	0	0	0	0	0	0	0	0	0000	01	00	Decodificación
2	0	0	0	1	0	0	0	1	0	0	0	0	0000	01	01	Ejecución de ADD
3	0	0	0	1	0	0	0	1	0	0	0	0	0001	01	01	Ejecución de SUB
4	0	0	0	1	0	0	0	1	0	0	0	0	0010	01	01	Ejecución de SLT
5	0	0	0	1	0	0	0	1	0	0	0	0	0011	01	01	Ejecución de SLTU
6	0	0	0	0	0	1	0	0	0	0	0	1	0000	01	10	Tercer ciclo de LW
7	0	1	0	1	0	0	0	1	0	1	1	0	0000	01	01	Ultimo ciclo de LW
8	0	0	0	0	0	1	1	0	0	0	0	1	0000	01	10	Tercer ciclo de SW
9	0	0	1	1	0	0	0	0	0	0	0	0	0000	01	01	Ultimo ciclo de SW
10	0	0	0	1	0	0	0	0	0	0	0	0	0000	01	01	Salto no cumplido
11	0	0	0	1	0	0	0	0	1	0	0	0	0000	01	01	Salto cumplido

2.7.3.- UCμ para ruta de Datos de ciclo corto



El Decodificador-Evaluador de la UCμ para la ruta de datos de ciclo corto, posee los siguientes valores internos:

		Número de μ instrucción		
IR.co	IR.func[5..0]	Si EVAL.out = 0	Si EVAL.out = 1	Comentario
000000	100000	000010	000010	Linea para ADD (μinst = 2)
000000	100010	000100	000100	Linea para SUB (μinst =4)
000000	101010	000110	000110	Linea para SLT (μinst =6)
000000	010010	001000	001000	Linea para SLTU (μinst =8)
100011		001010	001010	Linea para LW (μinst = 10)
101011		001101	001101	Linea para SW (μinst =13)
000100		001111	010000	Linea para BEQZ (μinst = 13 si salto no tomado; y μinst=16 si salto tomado)
000101		010000	001111	Linea para BNEZ (μinst = 16 si salto tomado; y μinst=15 si salto no tomado)

Las señales de control de 1 bit de **CC.μ** son las siguientes:

```
0.- MUXALU.sc      6.- DMAR.wr      12.- RA.wr
1.- MUXBRrd.sc     7.- IR.wr        13.- RB.wr
2.- MUXBRin.sc     8.- PC.wr        14.- RT.wr
3.- MUXPC.sc       9.- MD.wr        15.- MDRout.wr
4.- BR.wr          10.- MD.rd
5.- MDRin.wr       11.- MI.rd
```

El contenido de la Mμ para la ruta de datos de ciclo corto es la siguiente :

	Señales de control de 1 bit																			
μInst.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ALU.sc	SEX.sc	μMUX.sc	Comentario
0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0000	01	10	Fase de busqueda
1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0000	01	00	Decodificación
2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0000	01	10	Tercer ciclo de ADD
3	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0000	01	01	Ultimo ciclo de ADD
4	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0001	01	10	Tercer ciclo de SUB
5	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0000	01	01	Ultimo ciclo de SUB
6	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0010	01	10	Tercer ciclo de SLT
7	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0000	01	01	Ultimo ciclo de SLT
8	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0011	01	10	Tercer ciclo de SLTU
9	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0000	01	01	Ultimo ciclo de SLTU
10	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0000	01	10	Tercer ciclo de LW
11	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0000	01	10	Cuatro ciclo de LW
12	0	0	0	0	0	0	0	1	0	0	0	1	0	1	1	0	0000	01	01	Ultimo ciclo de LW
13	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0000	01	10	Tercer ciclo de SW
14	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0000	01	01	Ultimo ciclo de SW
15	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0000	01	01	Salto no cumplido
16	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0000	01	01	Salto cumplido

2.7.4.- Opción µprogramable

Los procesadores µprogramados de mCOVI son también µprogramables. Lo primero que se necesita es seleccionar una instrucción de la arquitectura DLXint que no esté incluida en el repertorio SubDLX (NOTA: que trate con números enteros), ver [HePa93,96].

Una vez seleccionada se debe escribir su µprograma intérprete. Para lo cual se aconseja obtener y estudiar previamente el µprograma subDLX a través de las opciones del menú UTILIDADES. El fichero generado servirá de guía para la realización del µcódigo del usuario.

En ese fichero existen dos partes: la primera contiene los datos que serán cargados en el Decodificador-Evaluador, y la segunda detalla el contenido de la Memoria de µcódigo.

El usuario deberá añadir una línea en la parte del DE con la información de la instrucción añadida : número de µinstrucción a ejecutar después de la decodificación. Por último, en la parte de la Mµ se deberán añadir tantas µinstrucciones como necesite el usuario para completar la instrucción.

El siguiente cuadro muestra el fichero necesario para implementar una instrucción de carga de memoria utilizando un direccionamiento modo *registro base más registro índice*. Para lo cual se ha "usurpado" el código de operación **XOR.** Este fichero (`xor_cl.mcd`) se encuentra en el directorio de µcódigo, y aparece en la página siguiente.

Las líneas añadidas se muestran en negrita y en cursiva. En este caso la µinstrucción de comienzo de ejecución es la número 12 (las anteriores están ya definidas), y se necesitan dos µinstrucciones para completar la ejecución de XOR.

Además de "inventar" nuevos comportamientos de instrucciones utilizando códigos de operación existentes en la arquitectura DLXint, se puede incorporar instrucciones que no incluye SubDLX y que con el µcódigo de usuario son perfectamente ejecutables. Un ejemplo claro de estas sería la de cualquier instrucción aritmética que soportara la ALU que hemos implementado, por ejemplo OR.

Al seleccionar un programa en ensamblador con instrucciones que no forman parte del repertorio SubDLX, mCOVI nos obliga indicar el fichero que contiene el µcódigo de usuario, para poder ejecutar dicho programa.

[HePa93] *ARQUITECTURA DE COMPUTADORES. Un enfoque cuantitativo*. J.L. Hennesy y D.A. Patterson, McGraw Hill 1993.

[HePa96] *COMPUTER ARCHITECTURE. A quantitive approach*. (Second Edition) J.L. Hennesy y D.A. Patterson, Morgan Kaufmann Publishers Inc 1996.

```
************************************************************
************************************************************
******* Microprograma para SUBDLX ciclo largo ***************
******* que implementa la instruccion XOR como carga de   ****
******* una palabra con formato registro/registro         ****
************************************************************
************************************************************
#
%% TABLA DE DECODIFICACION Y EVALUACION
#
# C.O.     FUNC     E=0      E=1
#
000000 ; 100000 ; 000010 ; 000010;  Líneas ya definidas
000000 ; 100010 ; 000011 ; 000011;
000000 ; 101010 ; 000100 ; 000100;
000000 ; 010010 ; 000101 ; 000101;
100011 ; 000000 ; 000110 ; 000110;
101011 ; 000000 ; 001000 ; 001000;
000100 ; 000000 ; 001010 ; 001011;
000101 ; 000000 ; 001011 ; 001010;
#
000000 ; 100110 ; 001100 ; 001100;  Línea para XOR  (µInst = 12)
#
#
%% MICROPROGRAMA
#
#  R.Datos        ALU   SEX  µMUX
#                  |     |    |
# 11               |     |    |
# 109876543210   ----   --   --
#
  100010000000 ; 0000 ; 01 ; 10;  µInstrucciones ya definidas
  000000000000 ; 0000 ; 01 ; 00;
  000100010000 ; 0000 ; 01 ; 01;
  000100010000 ; 0001 ; 01 ; 01;
  000100010000 ; 0010 ; 01 ; 01;
  000100010000 ; 0011 ; 01 ; 01;
  000001000001 ; 0000 ; 01 ; 10;
  010100010110 ; 0000 ; 01 ; 01;
  000001100001 ; 0000 ; 01 ; 10;
  001100000000 ; 0000 ; 01 ; 01;
  000100000000 ; 0000 ; 01 ; 01;
  000100001000 ; 0000 ; 01 ; 01;
#
  000001000000 ; 0000 ; 01 ; 10;  Microinstruccion 12 (Q2 de XOR)
  010100010100 ; 0000 ; 01 ; 01;  Microinstruccion 13 (Q3 de XOR)
```