Un ambiente para la enseñanza de la programación en lenguaje ensamblable

Enzo Chiariotti G. y David A. Koufaty A.
Departamento de Computación y
Tecnología de la Información
Universidad Simón Bolívar
Apartado 89000, Caracas 1080–A, Venezuela

Resumen

En este artículo se presentan las ventajas de simular arquitecturas para la enseñanza de la programación en lenguaje ensamblable y las características de un ambiente desarrollado para tal efecto. Este ambiente simula una arquitectura basada en el micro–procesador FUM diseñado en la Universidad Simón Bolívar. Esta arquitectura es particularmente apropiada para la docencia, ya que logra un buen compromiso entre ortogonalidad, poder expresivo y explotación del formato.

El ambiente provee además de la simulación de la arquitectura, el ensamblador de un lenguaje ensamblable definido para FUM, un enlazador y algunas herramientas para depuración.

1 Introducción

A pesar de que programar en Lenguaje Ensamblable se estila cada vez menos, es importante que el profesional de computación tenga conocimiento de los distintos niveles en la organización del computador, desde los niveles mas bajos de la lógica digital hasta los niveles superiores de los lenguajes de alto nivel.

Por otro lado, es bien conocida la relación 80–20, es decir, el 80% del tiempo de ejecución de los programas se invierte en el 20% del código de los mismos. En este marco la programación híbrida surge como una excelente alternativa para obtener una mejora sensible en la eficiencia del código generado, obteniendo una buena relación entre esfuerzo de programación y la velocidad de ejecución.

¿ Para qué tener un simulador ?

Existen variadas razones que dificultan la docencia de la programación en lenguaje ensamblable en arquitecturas reales. Aunque la base común de los lenguajes ensamblables es muy amplia, existen diferencias muy marcadas entre los diferentes lenguajes generados por las máquinas subyacentes. Estas máquinas tienden a ser complejas y presentan muchos detalles que agregan al proceso de aprendizaje un costo adicional. Por otro lado las más difundidas y económicas no son las mas simples y atractivas para la enseñanza.

Otro problema surge cuando se desea que el aprendizaje incluya una de las partes mas atractivas del mundo de la organización del computador: el manejo de excepciones; en estos casos es preferible diseñar simuladores de estos eventos, ya que de lo contrario los errores generados durante el aprendizaje pueden ser fatales para el resto de los usuarios en equipos multiusuarios y para la misma máquina. Esta perspectiva unida a la amplia difusión de los computadores personales genera una necesidad de desarollar simuladores de arquitecturas y lenguajes ensamblables.

Un simulador presenta varias ventajas respecto a las arquitecturas reales. La primera, y quizás mas importante, es su costo: si excluímos el costo de desarollo del simulador,

se pueden simular arquitecturas con el grado de complejidad deseado sobre máquinas bastantes económicas (por ejemplo computadores personales) disminuyendo el costo por equipo de laboratorio sensiblemente; incluso se extiende el alcance de los laboratorios, ya que hoy en día es común la presencia de computadores personales en el hogar, con lo cual los ejercicios pueden ser trabajados comodamente en la paz y quietud del mismo.

En segundo término, a través del simulador se puede obtener un conjunto de instrucciones sencillo y a la vez poderoso aunque no sea particularmente adecuado a las características reales de la máquina subyacente, logrando aislar los detalles innecesarios para el proceso de aprendizaje.

En tercer lugar, si el simulador es desarrollado sobre lenguajes de alto nivel, una vez programado es posible definir e implementar rápidamente diferentes máquinas que le den variadas alternativas y experiencias al estudiante. De igual forma se puede tener la misma arquitectura simulada sobre máquinas diferentes.

Por último un simulador de una arquitectura real muy costosa o de difícil acceso, trae la ventaja de poder usar el simulador para las primeras pruebas de los algoritmos y luego el recurso real para las pruebas definitivas y la versión en producción. De hecho hoy en día los diseñadores de máquinas paralelas ofrecen como primer elemento de prueba para los usuarios un simulador de la arquitectura sobre máquinas convencionales.

Como efecto de borde se produce la eliminación de los riesgos derivados de los errores en el aprendizaje, pues un error los programas de los estudiantes sólo afecta la arquitectura simulada y nunca a la máquina real, permitiendo que el producto sea robusto y resistente a las fallas de programación.

La principal desventaja de un simulador está en la velocidad de ejecución de los programas, el cual se ve seriamente disminuído. Sin embargo la velocidad de ejecución no es un factor determinante en la enseñanza.

2 El ambiente de programación FUM

FUM (*FUndamental Machine*) es un ambiente de programación diseñado en Septiembre de 1987 por los profesores Enzo Chiariotti, David Koufaty, Gustavo Lau y Alberto Torres en la Universidad Simón Bolívar. El proyecto surge como consecuencia de la crónica falta de equipo y de la flexibilidad que mostraba la idea de un simulador. Tradicionalmente el laboratorio de los cursos de Organización del Computador se desarrollaba sobre una PDP 11/45, la cual resultaba insuficiente para la demanda del curso.

En el proceso de diseño del ambiente fue necesario definir la arquitectura de la máquina, su lenguaje ensamblable y todos los elementos necesario para el desarrollo del programas (simulador de la arquitectura, ensamblador, enlazador y depurador).

2.1 La arquitectura FUM

FUM es un procesador tipo CISC (*Complex Instruction Set Computer*) inspirado fundamentalmente en dos computadoras, la PDP 11 y el Motorola MC68000. FUM se caracteriza por su ortogonalidad y la explotación del formato. Para presentar las características de la arquitectura a nivel de lenguaje de máquina convencional utilizaremos simultáneamente el lenguaje de máquina ensamblable estándar de FUM, FUMAL, descrito mas adelante.

1. Los buses de datos y de direcciones de FUM son de 16 *bits*, con lo cual el espacio direccionables es de 64K *bytes*. Sin embargo se pueden acceder operandos de 8, 16 y 32 *bytes*.

2. Posee 16 registros de 16 *bits* cada uno accesibles al usuario, de los cuales 8 son de datos (D0 – D7) y 8 de direcciones (A0 – A7). De estos últimos A6 corresponde con el apuntador a la pila o SP (*Stack Pointer*) y A7 al apuntador al programa o PC (*Program Counter*).

3. La codificación de los modos de direccionamiento se presenta siempre como un par modo-registro. FUM posee 8 modos de direccionamiento que actúan sobre uno de los grupos de registros, por lo cual la codificación de un operando requiere, excepto para algunas instrucciones, de 6 *bits*. Estos se resumen en la siguiente tabla:

Sintaxis	Modo	Registro	Nombre
Dn	0	n	Directo con registro de datos
An	1	n	Directo con registro de direcciones
@An	2	n	Indirecto con registro de direcciones
@An+	3	n	Indirecto con post-incremento
@-An	4	n	Indirecto con pre-decremento
x[An]	5	n	Indexado
@x[An]	6	n	Indexado indirecto
@@An+	7	n	Doble indirecto con post-incremento

La dirección efectiva de estos operandos se calcula fácilmente. Para los dos primeros es el registro correspondiente (Dn y An respectivamente). Para @An la dirección efectiva del operando viene dada por el contenido del registro de direcciones especificado, al igual que para el modo @An+, que además produce como efecto de borde el incremento posterior del registro. Lo mismo ocurre con @-An, excepto que el registro es decrementado antes de calcular la dirección efectiva. La dirección efectiva del modo indexado x[An] se obtiene sumando el contenido del registro de direcciones An y la constante x, mientras que en el indexado indirecto @x[An] se efectúa un indireccionamiento adicional. Por último, el modo @@An+ es similar al @An+ con un indirecionamiento adicional. La constante sumada o restada en los modos con post-incremento o pre-decremento varía de acuerdo al tamaño en *bytes* del operando, excepto para el modo 7 y para los registros A6 y A7, en cuyo caso es siempre 2.

4. FUM posee una palabra de estado de 16 *bits*, de los cuales sólo 8 son usados, con el siguiente formato:

0	0	0	0	P	P	P	0	0	0	0	T	N	Z	V	C

(a) *Bits* de prioridad PPP: representan la prioridad del procesador, que puede estar entre 0 y 7, siendo 0 la mínima. La prioridad es relevante para el esquema de entrada y salida del procesador, explicado más adelante.

(b) *Bits* de condición N, Z, V y C: reflejan el resultado de la última operación efectuada y corresponden a negativo, cero, desbordamiento (*overflow*), y acarreo (*carry*).

(c) *Trap* de *overflow* T: Habilita el *trap* automático de *overflow*.

5. El procesador identifica tres tipos de excepciones: los *traps* explícitos, los *traps* implícitos y las interrupciones. Para los primeros existen 8 instrucciones especiales, que generan el mismo número de *traps*, y el *trap* explícito de *overflow*. El *trap* de *overflow* se puede manipular además de manera implícita, habilitándolo o no por medio de intrucciones especiales. Las interrupciones son jerarquizadas por prioridades, donde cada dispositivo interrumpe enviando al procesador su prioridad, la cual es aceptada sólo si la prioridad del dispositivo es mayor que la del procesador. La prioridad del dispositivo llega al procesador por tres líneas, por lo cual la prioridad de los dispositivos está entre 1 y 7 (el 0 nunca produciría una interrupción).

   Las excepciones son atendidas empilando el PC y la palabra de estado, y cambiando el PC a la localidad indicada por los vectores de interrupción correspondiente al *trap* o interrupción. Cada vector es una localidad fija de memoria que debe contener la dirección de la rutina de servicio.

6. La entrada y salida es mapeada a memoria.

7. El conjunto de instrucciones está orientado fundamentalmente a la manipulación de operandos de 16 *bits*, aunque existen instrucciones para 8 *bits* y palabras largas de 32 *bits*. El conjunto incluye instrucciones para movimiento de datos, aritmética entera, operaciones lógicas, rotaciones y desplazamientos, control de flujo y del

micro-procesador. En el anexo se presenta una carta de ensamblaje con todo el conjunto de instrucciones. Como se observa el conjunto de instrucciones es bastante rico y flexible.

Las instrucciones requieren normalmente para su codificación de 16 *bits*, pero los modos indexado e indexado indirecto requieren de una palabra adicional. En el anexo se presenta la codificación de las instrucciones.

2.2 El micro-computador FUM-PC

El micro-computador FUM-PC implanta una computadora con una arquitectura d UNIBUS, que consta de un procesador FUM, una memoria de 32K *bytes* y 3 disposi tivos periféricos.

Los periféricos que se disponen son un teclado, un monitor y un temporizador. Ellos estan conectados al procesador a través de un PIC (*Programmable Interrupt Controller*) que envía las solicitudes de interrupción al procesador, asignando de manera estática las prioridades de los dispositivos; las solicitudes de los dispositivos con mayor prioridad son enviadas primero al procesador. Todos estos dispositivos son simulados y su efecto se presenta sobre los dispositivos reales de la máquina simuladora. Todos ellos pueden ser manipulados por medio de interrupciones y el teclado y el monitor además se pueden programar usando lazos de espera (*busy waiting*).

2.3 El lenguaje FUMAL

FUMAL (*FUndamental Machine Assembly Language*) es el lenguaje ensamblable estándar diseñado para FUM, y provee una notación simbólica conveniente para el lenguaje del procesador. Ademas de los modos nativos de la máquina se proveen cuatro modos de direccionamiento adicionales que son soportados por el ensamblador correspondiente. Estos son:

Sintaxis	Ensamblaje	Modo	Registro	Nombre
#sym	@PC+	3	7	Inmediato
sym	d(sym)[PC]	5	7	Directo relativo
@#sym	@@PC+	7	7	Directo absoluto
@sym	d(sym)[PC]	6	7	Indirecto

En los modos Inmediato y directo absoluto, la localidad siguiente a la instrucción contiene la constante sym, y d(sym) representa el desplazamiento de sym respecto a la localidad donde se colocará el mismo.

La dirección efectiva en el modo inmediato viene dado por la localidad donde se coloca la constante. En el modo directo relativo y el directo absoluto la dirección efectiva es la misma, la localidad sym; la diferencia entre ambas está en el ensamblaje y en el cálculo de la dirección efectiva a tiempo de ejecución, por lo cual el primero es adecuado para constantes relativas al comienzo del programa y el segundo para constantes absolutas. El indirecto efectúa un indireccionamiento a través de la dirección sym, de manera relativa.

2.4 Ambiente de programación

El ambiente está dirigido por menúes que facilitan la rápida integración del usuario. En el mismo se tiene acceso a la máquina virtual diseñada y a los utilitarios generales (ensamblador, enlazador, cargador y un depurador). El sistema posee dos niveles distintos: maestro y esclavo. El nivel del esclavo corresponde a la simulación de una tarjeta que posee un procesador FUM y 32 *Kbytes* de memoria RAM. Esta tarjeta es controlada por la máquina anfitriona a través de un reducido número de operaciones. El nivel de maestro corresponde a la máquina anfitriona que provee el *software* utilitario. Todos los procesos del ambiente corren en el maestro, mientras que en el esclavo sólo lo hacen los programas generados.

El ensamblador de FUMAL es un ensamblador estándar de dos pasadas, que permite entre otras cosas la resolución automática de referencias, cálculo de expresiones cons-

tantes a tiempo de ensamblaje y algunas pseudo–instrucciones para la manipulación de constantes y reservación de espacio. Además permite la referencia a símbolos definidos en otros módulos (símbolos externos) y la inclusión de archivos en el ensamblaje. El ensamblador genera dos archivos, uno con el código objeto y otro con el listado del ensamblaje del mismo y el valor de los símbolos.

El enlazador tiene interés por sí mismo, ya que con el uso de estructuras de datos adecuadas se logra realizar el enlace en una sola pasada sobre cada archivo objeto, sin necesidad de trabajo adicional. Este resuelve las referencias cruzadas y efectúa una relocalización parcial del código, generando un archivo ejecutable y un archivo con la resolución de las referencias cruzadas. La estructura del ejecutable y el objeto son similares, facilitando la creación futura de libreros.

En cuanto al resto de las herramientas, se dispone de un cargador que sólo efectúa la relocalización y copia del archivo ejecutable, y el depurador que provee facilidades para establecer múltiples puntos de parada, efectuar trazas y ejecutar paso a paso, además de permitir la modificación de registros y localidades de memoria.

El ambiente no provee un editor orientado al lenguaje pero el ambiente puede invocar un editor comercial elegido por el usuario.

Todas la características del sistema se presentan en detalle en [1, 2, 3].

3 Defectos y direcciones futuras

Desde la aparición de la primera versión en Septiembre de 1987 el ambiente ha experimentado una evolución bastante sostenida:

- 1.0 (Septiembre 1987) Primera versión.
- 2.0 (Octubre 1987) El ensamblador implementa el uso de símbolos en los modos de direccionamiento con el modo directo relativo (en la primera versión se utilizaba el

modo directo absoluto). El código objeto se representa en forma compacta (binario en lugar de ASCII) acelerando así la velocidad de carga.

- 3.0 (Noviembre 1987) Se añaden interrupciones y dispositivos periféricos. Se realizan pequeñas mejoras a la interfaz y al debugger.

- 4.0 (1988) El reconocedor de FUMAL se reescribió en Yacc [4]. Se introduce el cálculo de expresiones constantes a tiempo de ensamblaje y se modificó la notación de algunos modos de direccionamiento en FUMAL. Se aportaron mejoras significativas a la interfaz y a la documentación.

- 4.19 (Abril 1990) El sistema es sometido a una revisión completa. En el proceso de reingeniería el tamaño del código ejecutable se reduce de 102000 *bytes* a 53000 *bytes* y al mismo tiempo se aumenta la velocidad de ejecución. La simulación de los periféricos es desacoplada de la simulación del CPU. Se introdujeron en el lenguaje FUMAL las instrucciones JLO/CLO (*Jump/Call if lower*) y JHS/CHS (*Jump/Call if higher or same*), como sinónimos de JCS/CCS y JCC/CCC respectivamente. La documentación fué ampliada y actualizada.

- 4.40 (Septiembre 1990) Se introduce el enlazador y las directivas para inclusión de archivos y para la exportación e importación de símbolos.

- 4.41 (Enero 1991) Se agrega la generación de un archivo con la resolución de las referencias cruzadas durante el enlace.

El último avance significativo, el enlazador relocalizador, fué concluido en Agosto de 1990. En los últimos tres años las mejoras se han relacionado principalmente con la interfaz del ambiente y las herramientas de ayuda a la programación. Por otra parte el conjunto de instrucciones del procesador FUM se ha mantenido prácticamente inalterado desde su inicio.

El conjunto de instrucciones ha demostrado ser versátil. Sin embargo la experiencia ha puesto en evidencia la falta de algunas instrucciones. No es fácil escribir código

completamente independiente de la posición debido a la falta de una instrucción análoga intrucción a LEA del MC68000 (*Load Effective Address*): una instrucción que calcula la dirección efectiva de su operando. Las instrucciones de salto en los programas ocupan por lo general dos palabras, ya que no existen instrucciones que permitan saltos cortos relativos al PC. Las instrucciones MUL y DIV utilizan pares de registros, lo cual complica los algoritmos de asignación de registros en los compiladores.

Para el futuro ya se tiene diseñado un nuevo conjunto de instrucciones que incluye: TSB (*Test Byte*), LSW (*Load Status Word*), MEA (*Move Effective Address*), saltos cortos en una palabra (desplazamiento de 9 *bits* con signo) y cambio a la definición de MUL y DIV.

También se tiene previsto agregar un *bit* de modo protegido a la palabra de estado del procesador, lo cual va a permitir ilustrar algunos tópicos avanzados relacionados con sistemas de operación.

El ambiente de programación es susceptible de varias mejoras. Actualmente el ensamblador no permite definir macros. El depurador provee trazas, ejecución paso a paso y puntos de parada, pero no se dispone de ninguna facilidad para la depuración simbólica. Tampoco se tiene un desensamblador.

Actualmente se dispone de un prototipo de compilador de PASCAL realizado en el laboratorio de la cadena de lenguajes de programación. El uso del compilador permitir la realización de prácticas en programación híbrida. Sería también deseable disponer de compiladores de otros lenguajes de alto nivel.

El principal problema del computador FUM–PC es la falta de periféricos. Actualmente sólo se soportan en el ambiente una pantalla, un teclado y un temporizador. No existe un dispositivo de almacenamiento persistente. Se espera introducir la simulación de un disco y DMA en versiones futuras. Se está incluso considerando la posibilidad de permitir cierta configuración del computador virtual por parte del usuario, lo cual permitiría tener una familia de sistemas de cómputo basados en FUM.

La documentación del sistema es bastante completa: se dispone de manuales sobre el micro-procesador FUM, el lenguaje FUMAL, el ambiente de programación y el micro-computador FUM-PC. Sin embargo es necesario ampliarlos y documentar el código mismo del simulador.

Algunos defectos del ambiente tienen que ver con la naturaleza simulada del mismo. La simulación reduce la velocidad de ejecución considerablemente. El *software* del ambiente no corre sobre la máquina virtual misma y por razones obvias de eficiencia no sería deseable hacerlo; sin embargo la naturaleza esquizofrénica del sistema simulado puede inducir confusiones en el estudiante.

Otro problema debido a la simulación es la relación no realista entre la velocidad del procesador y la de los periféricos. Dispositivos como la pantalla son, en términos relativos, mucho más rápidos de lo que serían en la realidad. En la primera versión el procesador lograba realizar 10 accesos a la memoria (3 a 10 instrucciones) entre una interrupción de la pantalla y otra. A partir de la versión 4.19 el número ha sido aumentado sin disminuir la velocidad aparente de la pantalla. Sin embargo estas cifras distan mucho de la realidad. Mantener una relación cercana a la realidad en el simulador conllevaría una pantalla desesperadamente lenta.

Respecto a los últimos problemas mencionados no puede hacerse gran cosa por medio de *software*. Si se quieren solventar tales limitaciones hay que implementar un procesador FUM en hardware. Con el uso de una ULA comercial y la adopción de un esquema microprogramado sería posible implementar un procesador FUM en tiempo razonable y organizar un pequeño micro alrededor de él; pero esto requeriría un equipo de trabajo interdisciplinario y apoyo por parte de la institución.

4 Experiencias docentes

El desarrollo del ambiente permitió solventar cierta crisis en el laboratorio de Organización del Computador. Aumentó considerablemente la disponibilidad de tiempo de máquina del estudiante para la realización de sus proyectos. En general se percibe una mayor motivación por parte de los estudiantes hacia esta materia.

A través de la experiencia el lenguaje FUMAL a demostrado tener buenas cualidades desde el punto de vista académico. Es bastante simple sin que se pueda calificarlo como un juguete alejado de la realidad. El conjunto de instrucciones logra un buen compromiso entre ortogonalidad, aprovechamiento del formato y poder expresivo. El lenguaje FUMAL ha sido utilizado con éxito en la cadena de Lenguajes de Programación demostrando ser un ejemplo adecuado de lenguaje objetivo, tanto en la teoría como en el laboratorio del curso.

El *software* del ambiente ha servido también como material para proyectos de ingeniería de *software* de tamaño mediano, aptos para el laboratorio de la asignatura de Sistemas de Programas. Sin embargo las primeras experiencias de mejoras del ambiente con estudiantes durante el año 1989 fueron decepcionantes; no se pudo obtener un enlazador confiable y la evolución del sistema se atrasó considerablemente. Sentimos que esto se debió en gran parte a la falta de experiencia en la dirección de proyectos de este tamaño. Con la debida dirección, el enlazador fué implementado por un estudiante en una pasantía corta.

5 Conclusiones

A más de tres años de distancia del inicio de esta experiencia consideramos que el balance es francamente positivo. Los resultados obtenidos en el campo docente muestran que el esfuerzo invertido en el desarrollo del ambiente de programación FUMAL valió la pena.

Por otra parte ha demostrado su utilidad no sólo en los cursos de Organización del

Computador, ya que está siendo usado en los laboratorios de los cursos de compiladores como lenguaje objetivo y se plantea su uso en los cursos de Sistemas de Operación si se provee a FUM de modos de operación y protección de memoria.

En la actualidad se continúa el trabajo por dos vertientes. La primera consiste en evaluar las características del ambiente y del lenguaje para proveer las mejoras mencionadas. La segunda consiste en el desarrollo de simuladores de otras arquitecturas, ya que el relativo éxito de FUMAL sugiere la conveniencia de disponer de ambientes alternativos para tener distintos puntos de comparación. Actualmente se trabaja en la implementación de un ambiente similar pero basado en un procesador inspirado en la orientación RISC (*Reduced Instruction Set Computer*) y en la construcción de un simulador de una arquitectura micro-programable.

Agradecimientos

Deseamos agradecer a todos los estudiantes del curso de Organización del Computador que han acogido con gran entusiasmo el ambiente de programación FUMAL y han contribuído a la evolución del mismo aportando valiosas sugerencias. También deseamos agradecer a los estudiantes que han trabajado en la evolución del ambiente implementando nuevas características.

Un reconocimiento especial a Lisandro Ríos quien demostró una gran disposición al trabajo y logró uno de los objetivos largamente esperados.

Anexo: Carta de Ensamblaje de FUMAL

000000	NOP	000400	RTS	001000	STC
000100	EVT	000500	RTE	001100	CLC
000200	DVT	000600	HLT	00120n	STP <n>
000300	TPV	000700	RST	00130n	TRP <n>

0050dd	CLR <dst>	0054dd	TAS <dst>
0051dd	NEG <dst>	0055dd	SWP <dst>
0052dd	NOT <dst>	0056dd	SST <dst>
0053dd	TST <dst>	0057dd	ADC <dst>

0020dd	JMP <dst>	1020dd	CAL <dst>
0021dd	JEQ <dst>	1021dd	CEQ <dst>
0022dd	JGE <dst>	1022dd	CGE <dst>
0023dd	JGT <dst>	1023dd	CGT <dst>
0024dd	JCC <dst>	1024dd	CCC <dst>
0024dd	JHS <dst>	1024dd	CHS <dst>
0025dd	JVC <dst>	1025dd	CVC <dst>
0026dd	JPL <dst>	1026dd	CPL <dst>
0027dd	JHI <dst>	1027dd	CHI <dst>
0031dd	JNE <dst>	1031dd	CNE <dst>
0032dd	JLT <dst>	1032dd	CLT <dst>
0033dd	JLE <dst>	1033dd	CLE <dst>
0034dd	JCS <dst>	1034dd	CCS <dst>
0034dd	JLO <dst>	1034dd	CLO <dst>
0035dd	JVS <dst>	1035dd	CVS <dst>
0036dd	JMI <dst>	1036dd	CMI <dst>
0037dd	JLS <dst>	1037dd	CLS <dst>

004ndd	ADQ #<n>, <dst>	104ndd	SBQ #<n>, <dst>
020ndd	EQI Dn, <dst>	120ndd	XOR Dn, <dst>
021ndd	EQI #<n>, <dst>	121ndd	XOR #<n>, <dst>
022ndd	ASL Dn, <dst>	122ndd	ASR Dn, <dst>
023ndd	ASL #<n>, <dst>	123ndd	ASR #<n>, <dst>
024ndd	LSL Dn, <dst>	124ndd	LSR Dn, <dst>
025ndd	LSL #<n>, <dst>	125ndd	LSR #<n>, <dst>
026ndd	ROL Dn, <dst>	126ndd	ROR Dn, <dst>
027ndd	ROL #<n>, <dst>	127ndd	ROR #<n>, <dst>

03ssdd	AND <src>,<dst>	13ssdd	IOR <src>,<dst>
04ssdd	MOV <src>,<dst>	14ssdd	MOB <src>,<dst>
05ssdd	CMP <src>,<dst>	15ssdd	CMB <src>,<dst>
06ssdd	ADD <src>,<dst>	16ssdd	SUB <src>,<dst>
07ssdd	MUL <src>,<dst>	17ssdd	DIV <src>,<dst>

Referencias

[1] Enzo Chiariotti G. y David A. Koufaty A. El Procesador FUM. Reporte Técnico No. IT-1991-002, Departamento de Computación y Tecnología de la Información, Universidad Simón Bolívar, Caracas, Marzo 1991.

[2] Enzo Chiariotti G. y David A. Koufaty A. El Lenguaje FUMAL. Reporte Técnico No. IT-1991-003, Departamento de Computación y Tecnología de la Información, Universidad Simón Bolívar, Caracas, Marzo 1991.

[3] Enzo Chiariotti G. y David A. Koufaty A. Manual del Ambiente de Programación FUM. Reporte Técnico No. IT-1991-004, Departamento de Computación y Tecnología de la Información, Universidad Simón Bolívar, Caracas, Venezuela, Marzo 1991.

[4] Stephen C. Johnson. Yacc: Yet another compiler–compiler. Computing Science Technical Report No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.