# Trabajo Práctico Promoción
## Organización del Computador I

Test de la maquina Simulada SPARC

## Integrantes

De Simoni Luis Fernando LU:012/97
Revert Pablo LU:115/97



### Abstract

El objetivo de este informe es realizar un primer test de la maquina simulada SPARC bajo el modelo de simulación CELL-DEVS. EL mismo consiste en verificar la coherencia del modelo simulado, es decir para nuestro caso el realizar una verificación sobre la funcionalidad de las operaciones, verificando en este caso solo el resultado de las operaciones del set de instrucciones y el soporte brindado por la memoria.

Para el desarrollo del Testing en si hemos realizado sobre cada operación del set de instrucciones de la maquina SPARC varias simulaciones, contabilizando un total de 17100 simulaciones. Cada una de ellas independiente de la anterior generando cada una un resultado para poder su verificación.

### Rutina del test

El mecanismo utilizado para la simulación consiste en realizar alguna operación en particular y guardar su valor en memoria, luego se realiza un vuelco de la memoria obteniendo el estado posterior de la memoria a la ejecución de la instrucción. Se busca en este vuelco de memoria el valor esperado en la posición esperada de memoria, si este es el correcto la operación se ha realizado con éxito caso contrario se informa este evento.

## Desarrollo

Nuestro modelo de simulación cuenta con las siguientes instrucciones del Assembler SUN SPARC. Para estas se ha realizado un esquema de la simulaciones a realizar:

### *Test atómico de operadores*[1]

1. ST
   - 1.1. ST REG, [ADRRESS] — ∀ $REG_i$ (I IMPAR)
2. MOV
   - 2.1. MOV $REG_a$, $REG_b$ — ∀ $REG_a$, $REG_b$
   - 2.2. MOV CONST, REG — ∀ REG
   - 2.3. SET
   - 2.4. SET ETIQUETA, REG — ∀ REG
3. INC, DEC, SUB, ADD, SUBCC, ADDCC, Bxx
   - 3.1. INC CONST, REG — ∀ REG, CONST={0,FFFFFFFF,..}
   - 3.2. DEC CONST, REG — ∀ REG, CONST={0,FFFFFFFF,..}
   - 3.3. ADD REG, CONST, REG — ∀ REG, CONST={0,FFFFFFFF,..}
   - 3.4. ADD REG, REG, REG — ∀ REG, CONST={0,FFFFFFFF,..}
   - 3.5. ADDCC REG, CONST, REG — ∀ REG, CONST={0,FFFFFFFF,..}
   - 3.6. ADDCC REG, REG, REG — ∀ REG, CONST={0,FFFFFFFF,..}
   - 3.7. SUB REG, CONST, REG — ∀ REG, CONST={0,FFFFFFFF,..}
   - 3.8. SUB REG, REG, REG — ∀ REG, CONST={0,FFFFFFFF,..}
   - 3.9. SUBCC REG, CONST, REG — ∀ REG, CONST={0,FFFFFFFF,..}
   - 3.10. SUBCC REG, REG, REG — ∀ REG, CONST={0,FFFFFFFF,..}
   - 3.11. Bxx — ∀ Bxx, CONST1 {<,=,>} CONST2
     - 3.11.1. MOV CONST1, REG1
     - 3.11.2. MOV CONST2, REG2
     - 3.11.3. CMP REG1, REG2
     - 3.11.4. Bxx LABEL_SI
     - 3.11.5. LABEL_NO: MOV 0, REG3
     - 3.11.6. BA LABEL_FIN
     - 3.11.7. LABEL_SI: MOV FFFFFFFF, REG3
     - 3.11.8. LABEL_FIN: ST REG3, [RES]
4. LD, ST
   - 4.1. LD REG, [LABEL] — ∀ REG
   - 4.2. LD, ST — ∀ REG1, ∀ REG2, CONST en [0..n]
     - 4.2.1. LD REG1, [LABEL]
     - 4.2.2. MOV CONST, REG2
     - 4.2.3. INICIO: ST REG1, [LABEL], REG2
     - 4.2.4. LD REG1, [LABEL], REG2
     - 4.2.5. DEC REG2
     - 4.2.6. CMP REG2, %G0
     - 4.2.7. BNE INICIO
5. XOR, AND, OR, ANDN, XORcc, ANDcc, ORcc, ANDNcc
   - 5.1. XOR, AND, OR, ANDN, XORcc, ANDcc, ORcc, ANDNcc — ∀ Bxx, ∀ REG1, ∀ REG2 CONST1 {=, <>, <, >, = not } CONST2
     - 5.1.1. MOV REG1, CONST1
     - 5.1.2. MOV REG2, CONST2
     - 5.1.3. OP REG1, REG2

[1] En todos los casos exceptuando el **1**, se realizara luego de la operación tantos *ST REG, [RESULTADO n]* como registros hallan intervenido en la operación, para luego evaluar si la operación funciono correctamente.

5.1.4. Bxx LABEL_SI
5.1.5. LABEL_NO: MOV REG3, 0
5.1.6. BA LABEL_FIN
5.1.7. LABEL_SI: MOV REG3, FFFFFFFF
5.1.8. LABEL_FIN: ST REG3, [RES]

6. SLL, SRL, SRA ∀ REG1, ∀ REG2, CONST2 en [0..31]
   6.1.1. MOV CONST1, REG1
   6.1.2. OP REG1, CONST2, REG2
   6.1.3. ST REG1, [ORIGEN]
   6.1.4. ST REG2, [DESTINO]
7. MUL, MULScc, UMUL, UMULScc, DIV, DIVScc, UDIV, UDIVScc ∀ CONST1/2, REG1/2/3
   7.1.1. MOV CONST1, REG1
   7.1.2. MOV CONST2, REG2
   7.1.3. OP REG1, REG2, REG3
   7.1.4. ST REG1, [A]
   7.1.5. ST REG2, [B]
   7.1.6. ST REG3, [RESULT]

Para cada una de estas operaciones existen múltiples opciones, en cuanto a la utilización de ciertos registros o ciertos operandos inmediatos. Dado que el objetivo de la simulación no supone la correctitud de ningún medio de soporte, ni ninguna operación luego se hace necesario comenzar la verificación por los soportes básicos, es decir los registros.
El test de los registros es una parte fundamental, si no se asegura que cumplen con su especificación, nada se puede hacer en consecuencia. Para ello hemos realizado un test sobre operación SET, guardando en un registro un valor arbitrario y luego ese valor es escrito en memoria, por ultimo se verifica que el valor con el cual se deseaba realizar el SET es el que ha sido guardado en Memoria.

Este test fue exitoso en un 100%, se han realizado 99 simulaciones sobre registros aleatorios, siendo los inmediatos a almacenar números aleatorios también.

Nuevamente y suponiendo ya valido el soporte brindado por los registros podemos verificar su comportamiento al realizar la instrucción MOV *reg orig, reg dest*. Esta operación es implementada por el compilador como un ADD *reg orig , reg0 , reg dest*, siendo el registro reg0 equivalente a cero definido constante por la arquitectura. Es decir que en definitiva se realiza la verificación de esta operación. Para ello y ya que contamos con 32 registros se ha realizado una "posta" de un valor desde el registro 1 (set del valor reg1) hasta el registro 31 pasando por todos los registros, en todos los casos con un numero aleatorio como entrada.
El resultado es guardado en memoria en una dirección prefijada. Como resultado de este test hemos obtenido un 100% de resultados exitosos, al realizar 99 simulaciones.

Una vez que hemos verificado la operación MOV, continuaremos con operaciones simples como el INC y el DEC, estas operaciones son importantes ya que las utilizaremos por ejemplo para la verificación de los saltos condicionales, luego es imprescindible conocer su exactitud de cálculo. Para ello hemos realizado 279 simulaciones sobre cada una de estas operaciones obteniendo un 100% de aciertos, Se ha simulado estas operaciones con una constante inmediata como parámetro para todos los registros y dejando el resultado en un registro elegido al azar, por último este valor es guardado en memoria.

A esta altura ya podemos utilizar los DEC e INC y para setear los valores de la palabra de estado del procesador, de esta manera podremos verificar la correctitud de las operaciones de salto condicional.
En primer lugar verificaremos el BA, esta operación realiza un salto a una etiqueta siempre que es decodificada. Como resultado de este test, hemos obtenido un 100% de acierto en 279 simulaciones.

En el caso de la operación bn (salto siempre) se han obtenido resultados que no condicen con los que se debería obtener:

```
      !Test de BN, no salta nunca
      !Archivo: bnreg1.s
      !Los registros origen son aleatorios
      !Trabajo Promoción Testing:    De Simoni Luis Fernando
      !                               Pablo Revert

            set 82,  %r26     !82 en reg 26
            set 543854, %r1   !543854 en reg 1
            cmp %r26, %r1     !Comparo el reg26 con el reg1
            bn  LABELSI       !Guardo el resultado en memoria
      LABELNO: mov 0, %r10
             ba LABELFIN
      LABELSI: mov 4294967295, %r10
      LABELFIN: st %r10 , [dest]


               unimp

              .align 4
      valor:         .ascii "VALOR:"
      dest:      .word  FFFFFFFF  !Resultado del Test 1
```

**Archivo bnreg10r1.s utilizado para la verificacion de bn**

Al momento de realizar la verificacion de la operación se ha obtenido diferecias entre el valor esperado y el que se obtuvo. Generandose el archivo `bnreg10r1.dif`:

```
Campo  Tipo          Esperado            Encontrado          DIF
=====  ====          ========            ==========          ===
1      int32         0                   4294967295          ***
```

**Archivo bnreg10r1.dif generado por la verificacion de bn**

Analizando el ejemplo:

```
                  set 82,  %r26          !82 en reg 26
                  set 543854, %r1        !543854 en reg 1
                  cmp %r26, %r1          !Comparo el reg26 con el reg1
                  bn LABELSI    (1)      !Guardo el resultado en memoria
      LABELNO:    mov 0, %r10   (2)
                  ba LABELFIN   (3)
      LABELSI:    mov 4294967295, %r10
      LABELFIN:   st %r10, [dest] (4)
```

En la operación bn se decide no saltar, luego el PC apunta a (2) allí se setea el registro r10 con el valor 0, luego el PC apunta a (3) allí se hace provoca un salto obligado a (4), dado que la operación ba ha sido antes verificada obteniéndose valores correctos se supone que es correcto suponer que el PC apunta (4), de esta manera el valor a guardar en memoria es 0, y debería ser el valor esperado por el contrario se obtuvo en todos los casos 4294967295. Por ello se recomienda la verificación de esta operación

Para el respaldo de esta afirmación se han realizado 279 simulaciones. Generándose los correspondientes archivos .dif para cada uno de estos archivos.

Ahora podemos verificar de la misma manera las operaciones BEQ, que realizan un salto si los registros analizado son iguales. Nuevamente para esta prueba se han realizado 279 simulaciones para BEQ obteniéndose un 100% de acierto.

El error en las operaciones de salto se ha debido a no contemplar el salto retardado implementado por las arquitecturas SPARCs. Se debió incluir antes de la próxima operación luego del salto la operación NOP.

El archivo deberia ser:

```
      set 82,  %r26           !82 en reg 26
                  set 543854, %r1         !543854 en reg 1
                  cmp %r26, %r1           !Comparo el reg26 con el reg1
                  bn LABELSI    (1)       !Guardo el resultado en memoria
                  NOP
      LABELNO:    mov 0, %r10   (2)
                  ba LABELFIN   (3)
                  NOP
      LABELSI:    mov 4294967295, %r10
      LABELFIN:   st %r10, [dest] (4)
```

Nuevamente realizaremos un análisis mas profundo de la operación ADD, para ello realizamos 1519 simulaciones verificando el resultado obtenido logrando un 100% de éxito. Luego se han realizado 3069 simulaciones extras sobre esta operación.

Lo mismo se ha realizado con la operación ADDCC realizándose 258 simulaciones con resultados exitosos en todos los casos. En este caso se ha encontrado errores en la etapa de test, al generar estos archivos se incluyo como registros de trabajo al registro r0, de esta manera se intento sumar dos valores uno de ellos ha intentado guardarse en el registro r0.
Por ejemplo

```
      !Realiza la suma (CC) entre dos registros y guarda el resultado en memoria
      !Archivo: add2.s
      !Los registros origen son aleatorios
      !Trabajo Promocion Testing: De Simoni Luis Fernando
      !                     Pablo Revert

            set 9426270,  %r23      !9426270 en reg 23
            set 1848450, %r0  !1848450 en reg 0
            addcc %r0, %r23, %r15  !realizo la suma resultado en reg 15
            st   %r15, [dest]       !Guardo el resultado en memoria

                unimp

               .align 4
```

```
      valor:          .ascii "VALOR:"
      dest:       .word  FFFFFFFF  !Resultado del Test 2
```

**Archivo addccr15r2.s**

Obteniéndose el archivo de error:

```
Campo Tipo        Esperado          Encontrado        DIF
===== ====        ========          ==========        ===
1     int32       11274720          9426270                 ***
```

**Archivo addccr15r2.dif**

El valor esperado en este caso es el erróneo y el encontrado es el correcto, hemos decidido incluir este reporte dado que muestra que el registro 0 permanece inmutable y al ser utilizado en una suma (puede ser el caso del MOV también) el resultado esperado se corresponde con el resultado obtenido en una arquitectura SPARC, la cantidad de pruebas de este caso han sido 20, que han sido verificadas individualmente.

Lo mismo ha sucedido con la operación SUB, en este caso se han realizado 520 simulaciones obteniéndose un 100% de éxito. Se ha realizado una simulación extra de 2930 obteniéndose nuevamente valores correctos. Este esquema de análisis también se aplico a la operación SUBCC relizándose 532 simulaciones exitosas.

Nuestro próximo paso es verificar las operación de Shift de bits, tanto Shift Left como Shift Right, para ello se realizaron 151 y 279 simulaciones respectivamente obteniéndose en ambos casos un 100% de éxito.

Lo mismo se ha realizado para la operación XOR entre dos registros obteniéndose 100 % de éxito en 199 simulaciones. Se han realizado 984 simulaciones extras, contabilizando 1183 simulaciones exitosas sobre XOR. Este esquema ha sido aplicado a la operación AND en 636 simulaciones con los mismo resultados satisfactorios.

Por último hemos de pasar a operaciones mas complejas suponiendo la base de la simulación correcta. Comenzaremos verificando la instrucción SMUL, multiplicación con signo, nuevamente utilizaremos todos los registros en forma aleatoria y valores aleatorios.

En este caso se han realizado 712 simulaciones obteniéndose un 100 % de éxito.

Por último se ha realizado un test de la instrucción UDIV, división sin signo, se han realizado 1515 simulaciones, se han obtenido resultados

```
      !Test de UDIV
      !Archivo: udiv 10 100.s
      !Los registros origen son aleatorios
      !Trabajo Promoción Testing:   De Simoni Luis Fernando (ldesimon@dc.uba.ar)
      !                             Pablo Revert (prevert@bancorio.com.ar)

            set 274543375,  %r24    ! en reg 24
            set 13908050 , %r22
            udiv %r24, %r22 , %r10  !realiza una división
            st %r10 , [dest]        !Guardo el resultado en Memoria


                unimp
```

```
              .align 4
      valor:          .ascii "VALOR:"
      dest:       .word  FFFFFFFF  !Resultado del Test 100
```

**Archivo udiv10r100.s**

```
Campo Tipo        Esperado          Encontrado        DIF
===== ====        ========          ==========        ===
1     int32       19                1                 ***
```

**Archivo udiv10r100.dif**

En este ejemplo tenemos 274543375 dividido por 13908050, el resultado esperado es 19 pero se ha encontrado 1, esto ha sucedido en todas las simulaciones que se han realizado por ello el programa ha marcado OK a las simulaciones que como resultado obtienen 1.

## Deficiencias del Testing

Dado que la arquitectura posee múltiples registros y múltiples posibilidades de ingreso de parámetros para estas operaciones nos hemos limitado en esta simulación a casos aleatorios intentando así contemplar todo tipo de casos. Se han omitido por ejemplo las operaciones de STORE en memoria con transferencia de un Byte hacia la memoria en los casos testeados. Asimismo existen operaciones que no han sido verificadas con parámetros de entrada constantes inmediatas, ya que se ha preferido incluir implícitamente el movimiento y almacenamiento de los registros.
No se han verificado las operaciones de Punto Flotante y la verificación de instrucciones de la CACHE, como ser FLUSH. Asimismo se recomienda un analisis mas detallado de las operaciones UDIV, UDIVcc, SDIV, SDIVcc dado que los valores obtenidos no condicen con los esperados.

## Implementación

Todas las simulaciones se han realizado bajo el Sistema Operativo LINUX, sobre un procesador AMD K6II 450, con 64 Mb de Ram disponibles.
Se ha utilizado para la compilación de los archivos de código assembler, la implementación de las herramientas provistas por el paquete BINUTILS 2.9.5.0.24 realizándose la modificaciones oportunas para que estas generen código SPARC.
Para la ejecución de la simulación se ha utilizado un shell de comandos generados por Lex provisto por LINUX, se incluyen los fuentes de este parser.
Por ultimo se ha utilizado una aplicación desarrollada en Visual Basic para la verificación de los archivos de salida, esta aplicación realiza una verificación del valor esperado contra un archivo generado al momento de generar el código assembler, si se encuentra una diferencia se crea el archivo .dif con un análisis del valor esperado y el valor encontrado para su fácil análisis.
Este ultimo programa realiza un log sobre la simulación de tal operación obtendiendose un resumen de la cantidad de archivos verificados y los resultados obtenidos.

## Bibliografía

SPARC Assembly Language Reference Manual -Sun Microsystems (http://doc.sun.com)