Organización del Computador
Práctico 4

B. Gonzalez Kriegel - N. Wolovick

24 de abril de 2000

1. Un mapa de memoria tiene 2^{24} localizaciones direccionables. ¿Cuál es el menor ancho en bits que las direcciones pueden tener para poder referenciar toda la memoria?

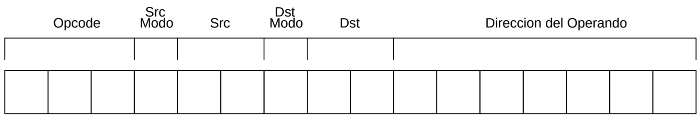
2. Dar la dirección de memoria más baja y más alta en un mapa de memoria direccionable por 20 bits, donde la menor unidad direccionable es una palabra de 4 bytes.

3. Escriba un fragmento de código ARC que efectúe el intercambio del contenido de las posiciones de memoria x y y. Use la menor cantidad de registros posible.

4. A continuación se muestra un pedazo de código assembler. ¿Qué es lo que hace? ¿Suma cosas o limpia una zona de la memoria? Comente la funcionalidad de cada una de las variables (`%r1,%r2,%r3,%r4`) y de las posiciones de memoria k, a, b y c, donde estas 3 últimas se suponen definidas en algún lugar del código. Una vez comprendida la funcionalidad del código, exprésela (tratando de mantener la estructura) en algún lenguaje imperativo de alto nivel y en algún lenguaje funcional a manera de especificación abstracta.

```
Y:      ld    [k], %r1
        addcc %r1, -4, %r1
        st    %r1, [k]
        bneg  X
        ld    [a], %r1, %r2
        ld    [b], %r1, %r3
        addcc %r2, %r3, %r4
        st    %r4, %r1, [c]
        ba Y
X:      jmpl  %r15 + 4, %r0
k:      40
```

5. Un pager de bolsillo contiene un pequeño procesador con 2^7 palabras de memoria de 8 bits. La ISA tiene 4 registros: `R0`, `R1`, `R2` y `R3`. Cada instrucción es una palabra de 16 bits (2 lugares de memoria), cuyo formato se muestra a continuación:



Los 3 primeros bits de la instrucción define el *opcode* y la siguiente tabla resume los 8 posibles códigos de operación:

Mnemónico	Opcode	Significado
LOAD	000	Dst ← Src o memoria
STORE	001	Dst o memoria ← Src
ADD	010	Dst ← Src + Dst
AND	011	Dst ← AND(Src,Dst)
BZERO	100	Salta si Src=0
JUMP	101	Salto incondicional
COMP	110	Dst ← complemento de Src
RSHIFT	111	Dst ← Src desplazado a la izq. 1 bit

Los 3 bits que definien la fuente o el destino responden a los siguientes patrones:

Modo	Patrón de bits
Registro	0
Directo	1

Registro	Patrón de bits
R0	00
R1	01
R2	10
R3	11

A lo más 1 operando puede obtenerse de modo directo (en una locación de memoria), y en caso de que alguno de los dos lo sea, el campo fuente o destino no se utiliza y la dirección de memoria se obtiene del campo dirección.

(a) Escriba un programa utilizando código objeto (no código ensamblador) que intercambie el contenido de los registros $R0$ y $R1$. Sólo utilice registros para no modificar la escasa memoria. No debería emplear más de 4 instrucciones, y es posible hacerlo en menos. Ponga 0's en los bits cuyo valor resulte indistinto.

(b) Escriba un programa en código objeto que intercambie el contenido de las posiciones de memoria 12 y 13. Como en el punto anterior, usted sólo podrá usar registros. Ponga 0's en los bits cuyo valor resulte indistinto.

6. Un programa escrito en leguaje ensamblador para la arquitectura ARC llama a la subrutina `foo`, pasándole 3 argumentos a, b y c. La subrutina tiene 2 variables locales, n y m y no retorna valor alguno.

   Usando la notación de libro muestre la posición del puntero de la pila y los elementos relevantes almacenados en ella para la convención de llamadas basada en pila en los siguientes momentos:

   (a) Justo antes de ejecutar la instrucción `call` en la etiqueta x.

   (b) Cuando se completa el marco de pila para la subrutina foo.

   (c) Exactamente antes de ejecutar la instrucción de carga en la etiqueta z.

```
              ⋮
!los argumentos a, b y c estan listos en la pila
x:      call  foo
z:      ld    %r1, %r2
              ⋮
              ⋮
foo:    !aqui empieza la subrutina
              ⋮
y:      jmpl  %r15 + 4, %r0
```

7. ¿Porqué `sethi` sólo carga los 22 bits altos de un registro? ¿No sería mejor cargar los 32 bits de un registro? Intente explicar este hecho.

8. Un programa compilado para un ISA de SPARC escribe el entero sin signo de 32 bits `0xABCDEF01` a un archivo y lo lee correctamente. El mismo programa compilado para un ISA de Pentium también trabaja de manera satisfactoria. Sin embargo cuando el archivo es transferido entre máquinas, el programa lee incorrectamente el entero del archivo como `0x01EFCDAB`. ¿Qué es lo que sucede?

9. Se tiene una ISA con opcodes de 8 bits, operandos y direcciones de 16 bits, donde los datos se mueven desde y hacia la memoria en trozos de 16 bits.

   (a) Escriba tres programas usando instrucciones de 3-direcciones, 2-direcciones y 1-dirección que computen la sentencia $A := (B+C)*(D+E)$. Su código no debe sobreescribir ninguno de los operandos y puede usar tantos registros ($R_0, R_1, \ldots$) para cálculos intermedios como sea necesario.

   (b) Compute el tamaño en bytes de cada uno de sus programas.

   (c) Compute el tráfico de memoria que cada uno de sus programa generará en tiempo de ejecución, incluyendo la obtención de cada instrucción (*instruction fetching*).

10. Repita el ejercicio anterior, sólo que ahora suponga la ISA de ARC para su programa. Utilice los mnemónicos `sub{cc}` para la resta y `smul` para la multiplicación.

11. Una arquitectura distinta a la ARC, que se clasifica como *de registros generales*, es la arquitectura basada en pila. Esta *stack-based architecture* (junto con la *acumulator architecture*) dominaron el mercado en los 60's y 70's, pero actualmente están en desuso.

    Por ejemplo, la siguiente secuencia de instrucciones evalúa la expresión $A+B*C$:

```
PUSH B
PUSH C
MULT
PUSH A
ADD
```

    Escriba un programa en esta arquitectura que evalúe la expresión $(B+C)*(D+E)$.

12. Cree la tabla de símbolos para el siguiente fragmento de código ARC. Sólo se pide que la tabla muestre el mapeo (símbolo, valor) y que se marque como "U" un símbolo no definido.

```
x        .equ  4000
         .org  2048
         ba    main
         .org  2072
main:    sethi x, %r2
         srl   %r2, 10, %r2
lab_4:   st    %r2, [k]
         addcc %r1, -1, %r1
foo:     st    %r1, [k]
         andcc %r1, %r1, %r0
         be    lab_5
         jmpl  %r15 + 4, %r0
cons:    .dwb  3
```

13. Traduzca el siguiente código ensamblador ARC en código objeto. Asuma que x está en la posición $(4096)_{10}$.

```
k        .equ  1024
               .
               .
               .
         addcc %r4 + k, %r4
         ld    %r14, %r5
         addcc %r14, -1, %r14
         st    %r5, [x]
```

14. Un **desensamblador** es un programa que efectúa la función inversa al ensamblador, es decir, este lee un código objeto y re-crea el código en lenguaje ensamblador. Dado el siguiente código objeto, desensámblelo en instrucciones del lenguaje de máquina ARC. Como no hay suficiente información en el código objeto para determinar los nombre de los símbolos, elija consecutivamente símbolos del alfabeto $\{a, \ldots, z\}$.

```
10000010 10000000 01100000 00000001
10000000 10010001 01000000 00000110
00000010 10000000 00000000 00000011
10001101 00110001 10100000 00001010
00010000 10111111 11111111 11111100
10000001 11000011 11100000 00000100
```

15. Cuando las macros `push` y `pop` son utilizadas, instrucciones innecesarias son insertadas cuando un `push` es seguido de manera inmediata por un `pop`. Expanda las macros e identifique que instrucciones resultan innecesarias.

```
        .begin
        .macro push arg1
        addcc  %r14, -4, %r14
        st     arg1, %r14
        .endmacro
        .macro pop arg1
        ld     %r14, arg1
        addcc  %r14, 4, %r14
        .endmacro

!  Comienzo del programa
        .org 2048
        pop    %r1
        push   %r2
        ⋮
        .end
```

16. El programa ARCSim soporta un conjunto más grande de instrucciones que las mostradas en el libro, como por ejemplo `nop`, `sra{cc}`, `halt` y `sub{cc}`. Aunque resulta más cómodo tenerlas como parte de la ISA, estas pueden ser reemplazadas fácilmente por instrucciones equivalentes y gracias al mecanismo de expansión de macros, podemos programar como si estas fueran parte de conjunto de instrucciones.
    (a) Muestre 3 instrucciones de distinto opcode para reemplazar `nop`.
    (b) De un conjunto de instrucciones que reemplacen `sub{cc}`.

17. Modifique el programa de la figura 4-14 del libro para que en vez de obtener la suma de los elementos de un arreglo, efectúe la división entera de cada elemento por 4, a la manera de un `map (/4)`. Los elementos del arreglo se consideran como enteros con signo.

18. Obtenga un programa en la ISA de ARC que calcule el $mcd(\%r1, \%r2)$, y que opere de manera similar al siguiente programa escrito en el lenguaje de los comandos guardados de Dijkstra:

$\{x = X \wedge y = Y\}$
do
    x>y $\rightarrow$ x:=x-y
  $\square$ y>x $\rightarrow$ y:=y-x
od
$\{x = y = mcd(X, Y)\}$

El registro $\%r1$ mantendrá la variable x y $\%r2$ la variable y. Utilice `subcc %r1, %r2, %r0` y `subcc %r2, %r1, %r0` para comparar los registros $\%r1$ y $\%r2$.

19. Modifique el programa anterior introduciendo un bloque de entrada y otro de salida, de manera tal que podamos llamar a la subrutina `mcd` utilizando la convención de *paso de parámetros en la pila*.

    Se busca que funcione el siguiente programa que hace uso de la rutina `mcd` e indica si dos números almacenados en las posiciones de memoria x e y resultan coprimos.

```
            !%sp .equ %r14 esta implicitamente definido

            .begin
            .org 2048
            ld     [x], %r1
            ld     [y], %r2
            addcc  %sp, -4, %sp
            st     %r1, %sp !push del 1er argumento
            addcc  %sp, -4, %sp
            st     %r2, %sp !push del 2do argumento
            call   mcd
            ld     %sp, %r3
            addcc  %sp, 4, %sp !pop del resultado
            addcc  %r3, -1, %r0 !%r3=1?
            be     isCoprime
            sethi 0, %r3
            ba cont
isCoprime:  sethi  0,%r3
            add %r3, 1, %r3 !el ser coprimo lo indico con %r3=1
cont:       !aqui continua el codigo
                   ⋮
mcd:        ⋮
            !Bloque de entrada
            ⋮
            !Codigo del ejercicio anterior
            ⋮
            !Bloque de salida
            ⋮
                   ⋮
x:          231
y:          22
            .end
```

    Puede resultar útil tener a mano las definiciones de `push` y `pop` dadas en el ejercicio 15.

20. ¿La ISA de ARC obliga a que `%r14` sea el puntero de la pila o esta es solamente una convención? Compare con la situación de `%r15`.

1 ARCSim y ARCAssembler en el laboratorio

En el laboratorio de computación está instalado todo lo necesario para ejecutar el simulador. Sólo basta tener activo el XServer y tipear desde una ventana de comandos

```
ARC
```

En `/usr/local/ARC` está el archivo `ARCTools.jar` que contiene todas las clases de ARCSim y ARCAssembler junto a varios archivos de documentación básica y el directorio `examples/` que justamente contiene ejemplos de códigos fuentes `.asm`.

El archivo `ARC` es un shell script que simplemente llama al JRE con los parámetros adecuados y está en `/usr/local/bin`.

El objetivo de utilizar el simulador es doble, pues por un lado permiten comprobar los programas codificados en los ejercicios, y por otra parte dan una idea más acabada de la ISA de ARC.

Los ejercicios que pueden ser codificados de manera más o menos directa son: 3, 4, 16, 17, 18 y especialmente el ejercicio 19.

Con las instrucciones que da el programa, más la información traducida y compilada que está disponible en la página de la materia bastaría para que se puedan codificar los programas. No obstante habrá un par de clases prácticas en el laboratorio donde se podrá consultar acerca del manejo de las ARCTools.