Organización del Computador
Práctico 5: Arquitectura del Conjunto de Instrucciones*

B. Gonzalez Kriegel - N. Wolovick

14 de mayo de 2001

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

2. A continuación se muestra una porción de código assembler. ¿Qué hace? Comente la funcionalidad de las variables `%r1,%r2,%r3,%r4` y de las posiciones de memoria `k, a, b, c`, donde estas 3 últimas se suponen definidas en algún lugar del código. Una vez comprendida la funcionalidad del código, escriba un programa que opere de la misma forma en algún lenguaje de alto nivel y también escriba la pre-post usando 1er orden y notación de arrays.

```
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
```

3. ¿Porqué `sethi` sólo carga los 22 bits altos de un registro? ¿No sería mejor cargar los 32 bits de un registro? Explique porque es así y como cargaría el valor `0xCAC0`=$(CAC0)_{16}$ en el registro `%r1`.

4. ! La *ARC* está a punto de ejecutar la instrucción `label: ba again`, es decir que el $\{PC = label\}$ y luego de la instrucción $\{PC = again\}$. Indique el rango[1] de memoria alcanzable por la instrucción de bifurcación.

5. La instruccion de la *ARCISA* `sethi 0,%r0` no tiene ningún efecto sobre el estado[2]. Una instrucción de este tipo se la denomina en casi todas las *ISA*s `nop` por *No OPeration*.

   (a) Muestre instrucciones simples que operan como `nop` usando saltos, operaciones aritméticas o lógicas y operaciones de carga.

   (b) ! ¿Porqué no podemos hacer `nop` con un `st`, `jmpl` o `call`?

6. ! ¿La *ARCISA* obliga a que `%r14` sea el puntero de la pila o esta es solamente una convención? Compare con la situación de `%r15`.

7. 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`. Explique que sucede.

*Los ejercicios marcados con ! son un poco más complicados, los marcados con * sirven para ganarse un punto por aterisco en los parciales y los marcados con + son para ganarse un punto por cruz en el examen final. No se puede acumular más de 2 puntos por persona por parcial o por final.

[1] Valor máximo y mínimo.

[2] Los registros y la memoria no cambian.

8. 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
```

9. Escribir las secuencias de 32 bits que definen las siguientes instrucciones:
`ld %r1,%r2`, `ld %r1,0,%r2`, `ld [%r1+0],%r2`.

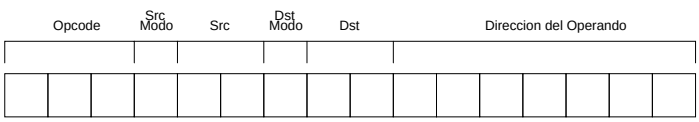
10. Traduzca el siguiente código ensamblador ARC en código objeto. Asuma que x está en la posición $(3072)_{10}$.

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

11. 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
```

12. 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[3]:

[3] Notar que $LOAD\ A, B \equiv STORE\ B, A$ y que los operandos no pueden ser directos al mismo tiempo.

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

Modo	Patrón
Registro	0
Directo	1

Registro	Patrón
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 en código ensamblador y 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.

13. 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. Notar que ésta es una optimización sólo aplicable a arquitecturas *RISC*.

```
        .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
```

14. 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 bloques de 16 bits.

(a) Escriba tres programas usando instrucciones de 3-direcciones, 2-direcciones y 1-dirección[4] que computen $A = (B+C)*(D+E)$ donde las variables $A \ldots E$ están en memoria. 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, tratando siempre de minimizar la cantidad de accesos a memoria.

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

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

[4] Arquitectura de acumulador.

15. En el ejercicio 14 utilizamos una *ISA* denominada *de registros generales*, donde cada operación podía contener 3,2 o 1 direcciones de memoria en sus operandos. Cuando tenemos 0 direcciones de memoria en los operandos, la denominamos *arquitectura de 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.

    A manera de 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 del ejercicio 14.

16. Se tiene el siguiente fragmento de código C, el cual ha sido transformado por algún compilador a un código *ARC*

```
for (i=0; i!=l; i++)
    A[i]=B[i]+C;
```

```
          .begin
A         .equ  1024            !direccion base del arreglo A
B         .equ  1424            !direccion base del arreglo B
C         .equ  1824            !escalar C
i         .equ  1828            !escalar i
l         .equ  1832            !escalar l

          .org  512
          st    %r0, [i]        !inicializacion del loop
          ba    endloop

loop:     ld    [i], %r1
          ld    [B+%r1], %r1    !toma B[i]

          ld    [C], %r2        !toma C

          add   %r1, %r2, %r2   !calcula la expresion B[i]+C

          ld    [i], %r1
          st    %r2, [A+%r1]    !almacena en A[i] la expresion

          ld    [i], %r1
          add   %r1, 4, %r1
          st    %r1, [i]        !incrementa el contador

endloop:  ld    [i], %r1
          ld    [l], %r2
          add   %r2, %r2, %r2   !ajusto la long.  a bytes
          subcc %r1, %r2, %r0
          bne   loop            !comprueba la condicion

          .end
```

    (a) Calcule $iCount(l)$, $ifMemcount(l)$ y $dataMemCount(l)$[5].

    (b) * Reescriba el programa de manera tal que los escalares l, i, C y las direcciones de los arreglos (pero no el arreglo en si) estén en registros del μprocesador. Vuelva a calcular las cantidades pedidas en el punto anterior.

    (c) + Explique porque el programa del punto anterior resulta óptimo respecto al *dataMemCount*. Proponga dos transformaciones de este programa que disminuya el $iCount(l)$ de manera significativa, es decir que la disminución sea proporcional al l.

[5] *iCount*:Cantidad de instrucciones ejecutadas. *ifMemcount*: Cantidad de palabras transferidas desde la memoria por *instruction fetching*. *dataMemCount*: Cantidad de palabras transferidas desde y hacia la memoria por movimiento de datos.

17. Obtenga un programa en la *ARCISA* 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. Compruebe su funcionamiento con ARCSim.

18. ! 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[6].

```
                !%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
```

[6] Puede resultar útil tener a mano las definiciones de `push` y `pop` dadas en el ejercicio 13.

19. + La tarea es comparar la eficiencia de memoria para cuatro estilos diferentes de arquitecturas del conjunto de instrucciones. Las arquitecturas son:

Acumulador Todas las operaciones son entre un registro dado y una locación de memoria.

Memoria-a-Memoria Los tres operandos de cada instrucción están en memoria.

Pila Todas las operaciones ocurren en el tope de la pila. Sólo las instrucciones `push` y `pop` acceden a la memoria; todas las otras operaciones sacan sus operandos de la pila y los reemplazan con el resultado. La implementación usa una pila para las dos entradas superiores; para acceder a otras posiciones del stack se usan referencias a memoria.

Carga-Almacenamiento Todas las operaciones ocurren en los registros y las instrucciones de registro-en-registro tienen tres operandos por instrucción. Hay 16 registros de propósito general, por lo que los especificadores de registros son de 4 bits.

Para medir la eficiencia de memoria, asuma lo siguiente para los cuatro conjuntos de instrucciones:

- El *opcode* siempre es de 1 byte (8 bits).
- Las direcciones de memoria son de 2 bytes (16 bits).
- Todos los datos de operandos son de 4 bytes (32 bits).
- Todas las instrucciones tienen una cantidad entera de bytes de largo.

No hay otras optimizaciones para reducir el tráfico de memoria, y las variables $A \ldots D$ están inicialmente en memoria.

Invente sus propios mnemónicos de lenguaje ensamblador y escriba el mejor código ensamblador equivalente para el siguente fragmento de lenguaje de alto nivel. Escriba las cuatro secuencias de código para:

```
A = B + C;
B = A + C;
D = A - B;
```

(a) Calcule *ifMemCount* y *dataMemCount* en bytes para cada una de las secuencias.

(b) ¿Cuál es la arquitectura más eficiente medida en tamaño de código? ¿Cuál respecto al ancho de banda total de memoria (instrucciones+datos)?