Facundo Martinez Abeldaño 224/19

Parcial Organización del computador

Ejercicio 1 Sabiendo que `a1 = 0xffffffff`, ¿cuánto queda almacenado en `a2` luego de realizar la operación: `andi a2,a1,0xf00`?

Solución:

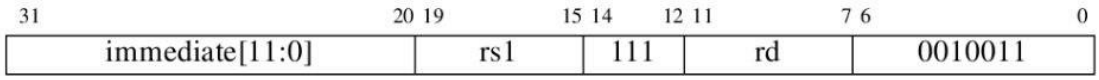
Buscamos en el libro de RISC-V que hace la instrucción andi (and de dos operandos inmediatos)

andi rd, rs1, immediate `x[rd] = x[rs1] & sext(immediate)`

AND Immediate. Tipo I, RV32I y RV64I.

Calcula el ANDa nivel de bits del *inmediato* sign-extended y el registro x[*rs1*] y escribe el resultado en x[*rd*].

Forma comprimida: **c.andi** rd, imm



a2 = 0xffffffff & singext(0xf00) = 0xffffffff & 0xfffff00 = 0xfffff00

Haciendo el and Bitwise obtenemos 0xfffff00

Ejercicio 2 Escribir una función que calcule (y devuelva) la suma de los primeros n números naturales. n es el único argumento de la función, que tiene que respetar la convención de llamadas de RISC-V

Solucion:

Buscamos en el libro de RISC-V como es la convención de llamadas de RISC-V

Facundo Martinez Abeldaño 224/19

Registro	Nombre ABI	Descripción	¿Preservado en llamadas?
x0	zero	Alambrado a cero	—
x1	ra	Dirección de retorno	No
x2	sp	Stack pointer	Sí
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Link register temporal/alterno	No
x6–7	t1–2	Temporales	No
x8	s0/fp	Saved register/frame pointer	Sí
x9	s1	Saved register	Sí
x10–11	a0–1	Argumentos de función/valores de retorno	No
x12–17	a2–7	Argumentos de función	No
x18–27	s2–11	Saved registers	Sí
x28–31	t3–6	Temporales	No
f0–7	ft0–7	Temporales, FP	No
f8–9	fs0–1	Saved registers, FP	Sí
f10–11	fa0–1	Argumentos/valores de retorno, FP	No
f12–17	fa2–7	Argumentos, FP	No
f18–27	fs2–11	Saved registers, FP	Sí
f28–31	ft8–11	Temporales, FP	No

Usamos los registros con la convención y la intención que indica este cuadro

a1 va a ser el resultado que quiero retornar

t1 mi contador, (registro temporal)

a0 es el valor donde va a estar guardado n pues es para argumentos de función y variables de entorno

```
37  def sumarn(n):
38      res = 0
39      i = 0
40      while(i<=n):
41          res = res + i
42          i = i + 1
43      return res
44
45  ASM
46
47  subrutina_sumar_n:
48                      add t1 , zero , zero  #limpio el registro t1 = 0 + 0
49                      add a1 , zero , zero  #limpio el registro a1 = 0 + 0
50                      beq a1 , a0 , volver  #si a0 = a1 = 0 => la suma da cero y retorno, sino caigo al loop
51
52  loop:
53                      add  a1, a1, t1       # a1 = a1 + t1
54                      addi t1, t1, 0x01     # t1 = t1 + 1
55                      bge  t1, a0, volver   # si t1 >= n => salto a volver (ret)
56                      j loop                # si not(t1 >= a0) => t1 < n y loopeo
57
58  volver:
59                      ret                   # retorno a la rutina principal
60
```

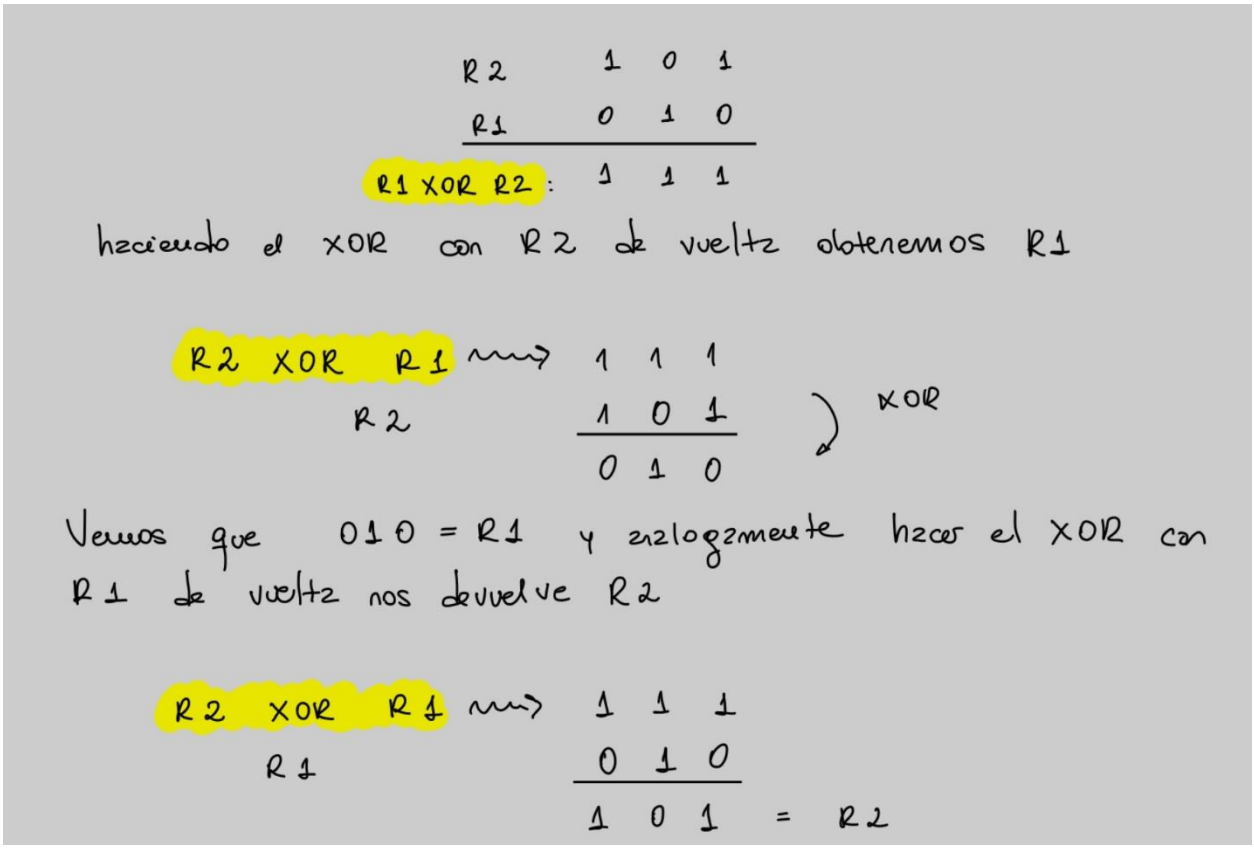
Facundo Martinez Abeldaño 224/19

Durante la corrección también se hablo de porque se pudiera haber optimizado el ejercicio anterior (sacando un registro) y la utilidad de usar un registro menos para temas de performance (no ir a buscar al registro el valor, setearlo, etc.) y se hizo especial énfasis en no usar pseudo-instrucciones. En esta nueva solución arreglamos eso limpiando el registro a1 sumando 0+0 y guardandolo

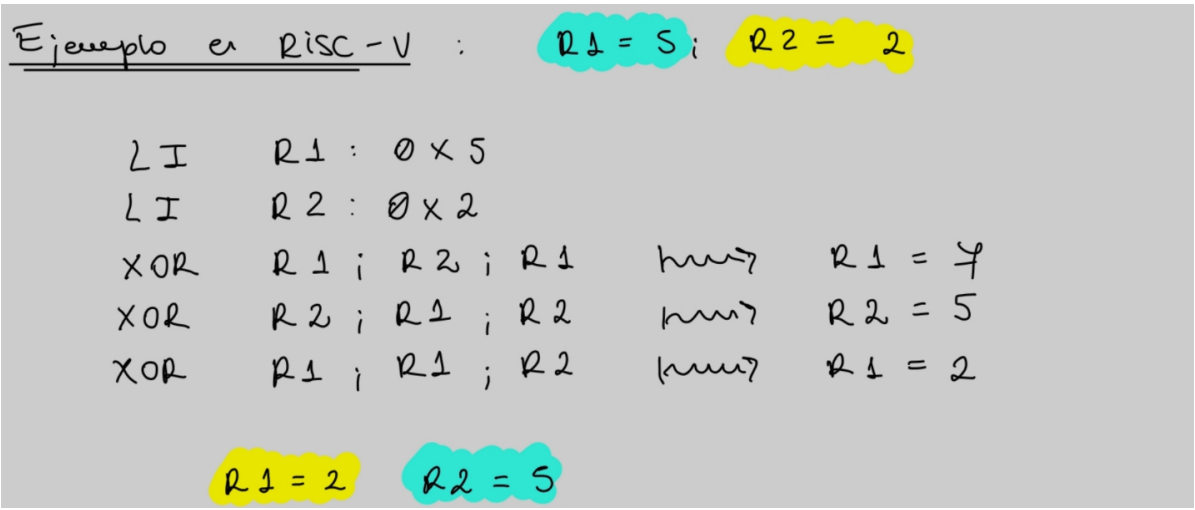
Ejercicio 3 Dados dos registros, mostrar la forma de intercambiarlos sin la intervención de un tercero.

Solución: Haciendo un XOR entre dos registros se puede intercambiar uno por otro hagamos un ejemplo simple para ver como funciona

Tomamos R1 = 010 , R2 = 101 quiero ver que haciendo el xor bitwise R2 = 010 y R1 = 101

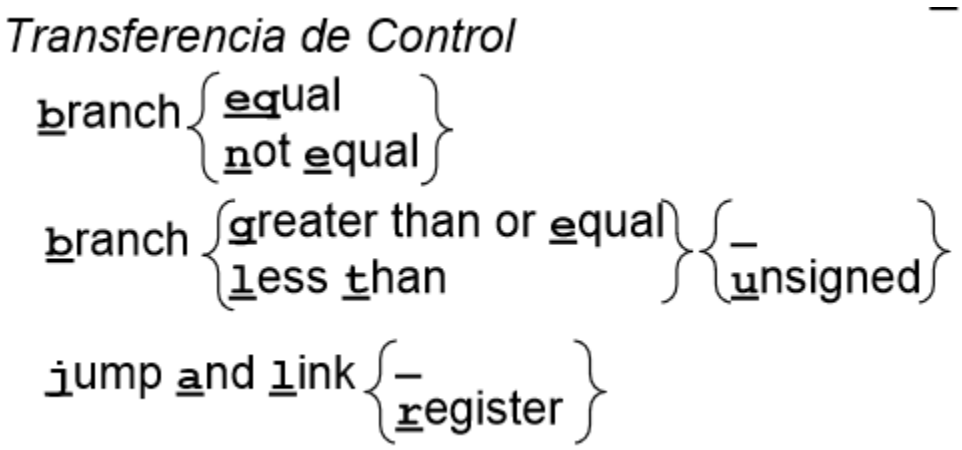


Una posible solución para intercambiar dos registros sin usar un tercero (usando este mismo ejemplo) seria hacer este procedimiento en RISC V



Ejercicio 4 ¿Cómo se resuelve la lógica de control (`branching`)? ¿Qué similitudes y/o diferencias existen con la máquina Orga1?

Solución: La lógica de control se realiza con los operadores jump y branch



Las instrucciones jump funcionan igual que Orga 1 (jumps no condicionales) en su mecanismo, toman el PC y lo modifican en base al valor de la etiqueta (calcula el offset y lo suma al PC antes extendiéndole el signo)

Hay jumps que no se encuentran en la maquina Orga 1 (JAL, JALR, JR)
Cada uno de estos saltos involucra la manipulación de un registro ya sea ponerlo en un registro y saltar (extendiendo el signo y sumando el valor del registro) (JALR), saltar a una dirección de memoria de un registro (JR), o guardar el PC en un registro y saltar (JL)

Por otro lado, las branches son similares los JN, JLE, JE de Orga 1 pero no son iguales en su mecanismo, si bien en estas instrucciones (las branches) hacen la operación de calcular el desplazamiento de la etiqueta y sumárselo al PC en Orga 1 esto no sucede, en Orga 1 los saltos se deciden en base a el estado de las flags que haya dejado la operación anterior mientras que en RISC-V la instrucción resuelve si saltar o no sin consultar el estado de las flags de la operación anterior.

Por ejemplo nosotros en Orga 1 para saber si un registro es más chico que otro primero haríamos la operación SUB R1;R2 ; luego haríamos un JLE a la etiqueta que deseamos y si las flags se portan bien saltaríamos, esto lo podemos hacer en RISC-V simplemente haciendo un BLE R1,R2

Ambas instrucciones (branches y jumps) manejan los saltos extendiendo el signo de los bits de la dirección, los multiplica por 2 y la suma al PC
En el caso de los branches ie.
PC = PC + Sing-ext(Imm(12bits) x 2)

En el caso de los Jumps ie. JAL
PC = PC + Sing-ext(Imm(20bits)x2)

Ejercicio 5 ¿A qué se llama **heap**? ¿A qué se conoce como **heap overflow**?

Solucion

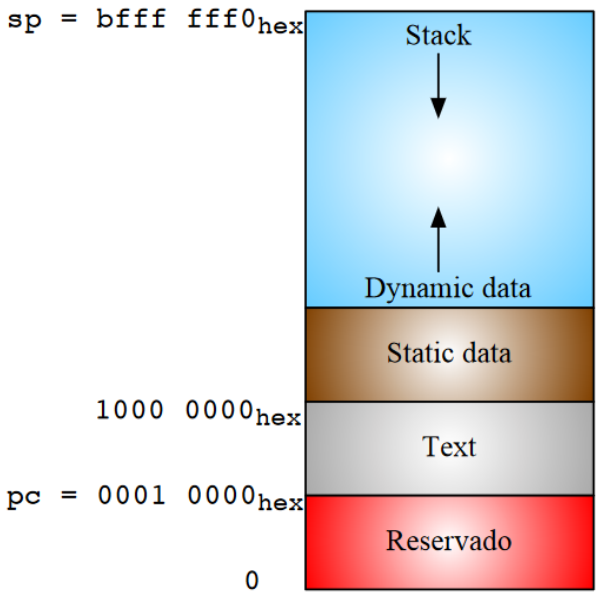


Figura 3.10: Reserva de memoria para el programa y datos en RV32I. Las direcciones más altas aparecen en la parte superior de la figura y las direcciones más bajas en la parte inferior. En esta convención del software de RISC-V, el stack pointer (sp) comienza en bfff fff0$_{hex}$ y crece hacia abajo hacia el área de *Static data*. El área de *text* (código del programa) comienza en 0001 0000$_{hex}$ e incluye las librerías *linkeadas* estáticamente. El área de *Static data* comienza inmediatamente después del área de *text*; en este ejemplo, asumimos que es la dirección 1000 0000$_{hex}$. Los datos dinámicos, reservados en C usando `malloc()`, están justo después del *Static data*. Llamado el `heap`, crece hacia el área de stack e incluye las librerías *linkeadas* dinámicamente.

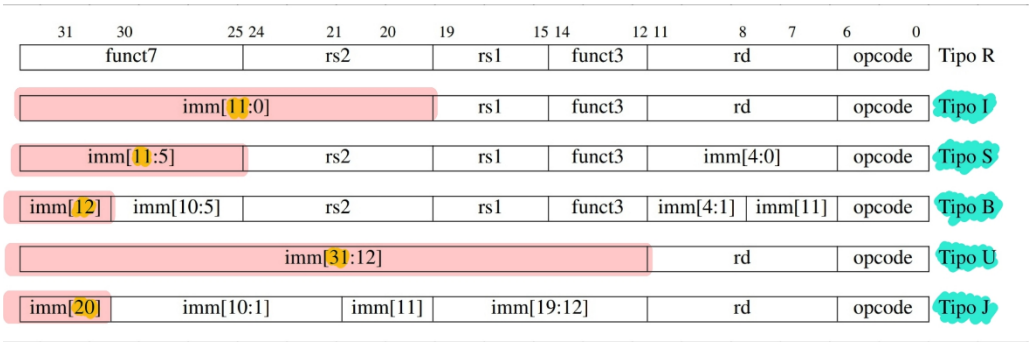
Facundo Martinez Abeldaño 224/19

El *heap* es un espacio de memoria reservado para la localización de memoria dinámica es usado para almacenar datos en ejecución y como bien dice la foto crece hacia el área del stack, el heap overflow se produce cuando esa memoria se desborda y colisiona con el stack reescribiendo así registros reservados de cada una.

Ejercicio 6 ¿En qué posición se encuentra el bit más significativo del campo inmediato dentro de la instrucción? ¿Depende del tipo de instrucción o de la instrucción en sí? ¿Por qué cree que fue diseñado así el formato de instrucción?

Solución:

El bit más significativo **del campo inmediato** se encuentra **siempre** al final de la instrucción que lo necesite (tipos I, S, B, U, J)



el bit mas significativo marcado en amarillo y el final de la instrucción marcado en rojo, en verde cada instrucción que necesita de un valor inmediato

En el libro de RISC esta escrito que el el bit mas significativo de la instrucción esta esta al final para que la extensión de signo del inmediato pueda continuar antes de la codificación (pagina 19) al principio