Ayudantía 5:

RISC-V & RARS

Andrés González

Tomás Contreras

Rocío Hernández

003-1040559 1250 003-77156.8 1760 0009-14563.7 73273

Temas

- Emulador RARS
- Registros
- Assembly RISC-V
- Programación en assembly RISC-V

01
02
03
04
05
06

RARS

- Risc Assembler, Runtime and Simulator.
- Requiere Java 8 o superior (si corren Minecraft, corren el emulador).
- Soporta RISC-V IMFDN Base (riscv32 and riscv64), i.e. todo lo que necesiten.
- Nosotros usaremos registros de 32 bits.

Pasos que emula RARS

Assembly language 1

Programa en texto plano en assembly

Assembler 2

Produce un módulo de objeto en lenguaje de máquina

Linker 3

Combina múltiples módulos con las librerías correspondientes y produce un ejecutable en lenguaje de máquina

Loader 4

Ubica el ejecutable en las direcciones de memoria apropiadas para que sea ejecutado por el procesador.

003-1040559 1250 003-77156.8 1760 0009-14563.7 73273

Registros

La extensión base RV32I considera 32 registros para datos, de 32 bits cada uno, desde x0 a x31.

x0 es un hard-wired zero -> siempre son 32 ceros, intentar cargarle algo no hace nada.

Además existe un duplicado de otros 32 registros para punto flotante, f0 a f31, que son los que se deben usar cuando se opera con números de punto flotante, que son parte de las extensiones F y D.

Los registros designados para el cero (x0/zero), return address (x1/ra), stack pointer (x2/sp), global pointer (x3/gp) y thread pointer (x4/tp) están sólo en la sección para enteros, esto significa que tenemos más registros libres para floats que para ints.

La convención de llamada (para que sirve cada registro y su alias) está especificada en el green card de RISC-V.

Elementos básicos para un programa:

Template

Es la estructura básica que tiene nuestro programa.

Load/Store y mov

Nos permite leer y escribir entre memoria y registros.

Aritmética

Con esto podemos sumar, restar, multiplicar y hacer cálculos en general.

Saltos

Permiten ir a labels particulares sin pasar por todo el programa.

Comparaciones

A través de comparaciones entre elementos podemos implementar predicados lógicos y loops.

Bitwise y lógica

Nos permiten manipular directamente los bits de un registro.

003-1040559 1250 003-77156.8 1760 0009-14563.7 73273

Template

Un programa en assembly RISC-V suele tener dos macrosecciones:

- .globl: se provee el label de inicio para el linker.

1) .data: donde se declaran elementos que estarán almacenados en memoria.
2) .text: el código del programa a ejecutar.

El orden de estas no es relevante, podría ser .data > .text > .globl o .text > .globl > .data, pero preferimos el formato anterior porque es más fácil de leer.

Template

```
.globl __start

.data

     A: .word 12354

     B: .byte 0xFF, 0xC4,

.text

     __start:

          # do stuff
```

.data

.ascii	string sin null terminator
.asciz	string CON null terminator
.byte	8-bit value(s)
.double	Double precision fp (64b)
.dword	64-bit word
.eqv	Sustituye un operando por un símbolo (alias)
.float	Single precision fp (32b)

.globl	Referencia para el linker
.half	16-bit halfword
.string	.asciz alias
.word	32-bit word
.zero	zero bytes

Instrucciones en RISC-V

Todas las instrucciones son de 32 bits de largo, pero existen extensiones para 16 bits, que no usaremos.

Como las instrucciones tienen un largo fijo, se utilizan distintos formatos: R, I, S, SB, U y UJ en para 32 bits.

Ojo que las green cards que encuentren en internet son sólo una referencia y podrían faltar/tener más cosas, dependiendo de las extensiones. Las instrucciones relevantes (y que funcionan) son las que aparecen en la página de ayuda de RARS.

w(゜Д゜)w

Load/Store y mov

Load/Store: Son instrucciones que permiten referenciar o dereferenciar registros a memoria o cargar/descargar registros.

Mov permite mover el contenido de un registro a otro registro.

lb, la, lbu, lui, lw, ...

sw, sb, ...

01
02
03
04
05
06

Aritmética

Tenemos instrucciones para sumar, restar, multiplicar, dividir, obtener el resto, max/min, etc.

Además varias de estas tiene variaciones para especificar si se quiere tratar con números con o sin signo.

Las instrucciones para trabajar con enteros son distintas de las de punto flotante, pero casi todas tienen su equivalente.

add, addi, sub, mul, div, rem, ...

01
02
03
04
05
06

Saltos

Es posible hacer saltos incondicionales como también se pueden hacer saltos ligados a cierta dirección de retorno.

j, jal, jalr, jr

Todas las instrucciones de salto requieren de un label al que saltar.

Branching

Funcionan como un salto condicional, en que se decide si se debe pasar a un label en particular dada una comparación sobre dos registros.

Si quieren tomar una decisión sobre más de un registro, las comparaciones se deben hacer por separado.

beq, bge, bgeu, blt, bltu, bne, ...

Operaciones bitwise: OR & ORI

Recibe dos patrones de bits del mismo largo y ejecuta el or lógico entre cada bit correspondiente.

```
.data

     A: .byte 5, 19

.text

     la t1, A

     lbu a1, 0(t1)

     lbu a2 1(t1)

     or a0, a2, a1
```

```
a1 -> 00000101
a2 -> 00010011
or
a0 == 00010111
```

Operaciones bitwise: OR & ORI

Recibe dos patrones de bits del mismo largo y ejecuta el or lógico entre cada bit correspondiente.

```
.data

     A: .byte 19

.text

     la t2, A

     lbu a1, 0(t1)

     ori a0, a1, 7
```

```
x2   -> 00010011
7    -> 00000111
ori
a0   == 00010111
```

Operaciones bitwise: XOR & XORI

Recibe dos patrones de bits del mismo largo y ejecuta el xor lógico entre cada bit correspondiente.

```
.data

     A: .byte 24, 17

.text

     la t1, A

     lbu a1, 0(t1)

     lbu a2, 1(t1)

     xor a0, a1, a2
```

```
a1  -> 00011000
a2  -> 00001001
xor
a0  == 00010001
```

Operaciones bitwise: XOR & XORI

Recibe dos patrones de bits del mismo largo y ejecuta el xor lógico entre cada bit correspondiente.

```
.data

     A: .byte 24

.text

     la t1, A

     lbu a1, 0(t1)

     xori a0, a1, 43
```

```
a1    -> 00011000
43    -> 00101011
xori
a0    == 00110011
```

Operaciones bitwise: AND & ANDI

Recibe dos patrones de bits del mismo largo y ejecuta el and lógico entre cada bit correspondiente.

```
.data

     A: .byte 5

     B: .byte 4

.text

     la a1, A

     la a2, B

     lbu a3, 0(a1)

     lbu a4, 0(a2)

     and a5, a3, a4
```

```
a3  -> 00000101
a4  -> 00000100
and
a5  == 00000100
```

Operaciones bitwise: AND & ANDI

Recibe dos patrones de bits del mismo largo y ejecuta el and lógico entre cada bit correspondiente.

```
.data

     A: .byte 5

.text

     la a1, A

     lbu a3, 0(a1)

     andi a5, a3, 17
```

```
a3   -> 00000101
17   -> 00000100
andi
a5   == 00010110
```

Operaciones bitwise: NOT

not es una pseudo instrucción, i.e. no es soportada por hardware, pero el assembler se encarga de desenrrollarla.

Se implementa usando xori

```
.data

     A: .byte 7, 8

.text

     la t1, A

     lbu a1, 0(t1)

     lbu a2, 1(t1)

     not a1, a2
```

```
a1  -> 00000111
a2  -> 00001000
not
a1  == 11110111
```

Operaciones bitwise: Shifts Left y Right

Objetivo:

Cambiar de 0000 0000 0000 1101 en dirección de memoria x19

hacia 0000 0000 1101 0000 en dirección de memoria x11

Operaciones bitwise: Shifts Left y Right

Objetivo:

Cambiar de 0000 0000 0000 1101 en dirección de memoria x19

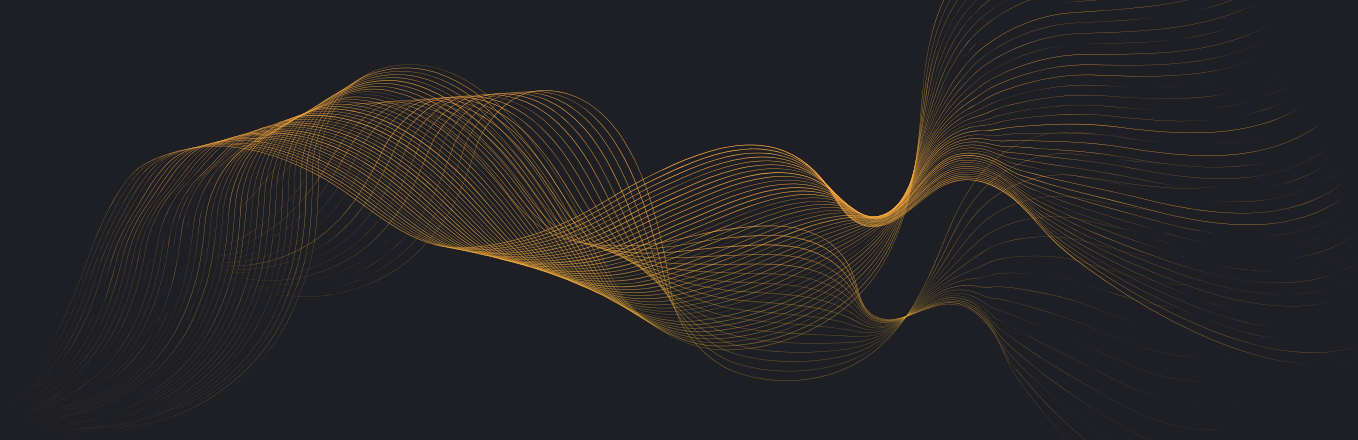
hacia 0000 0000 1101 0000 en dirección de memoria x11

```
slli x11, x19, 4
```

donde se guarda

el que utilizamos

cuanto lo movemos en bits



CODING TIME!

q(≧▽≦q)

003-1040559 1250 003-77156.8 1760 0009-14563.7 73273