DCC

DEPARTAMENTO DE CIENCIA
DE LA COMPUTACIÓN

IIC2343

Arquitectura de Computadores

Clase 9 - Arquitectura RISC-V

Profesor: Germán Leandro Contreras Sagredo

Objetivos de la clase

- Conocer la arquitectura RISC-V a nivel general.
- Conocer el *set* de instrucciones de la ISA RISC-V.
- Conocer las convenciones de llamada utilizadas en RISC-V.
- Ver ejemplos de código en clases a través del simulador RARS.

Hasta ahora...

- Ya conocemos la arquitectura x86 con mayor profundidad, su evolución y la ISA de 16 bits.

- A su vez, aprendimos sobre la convención de llamadas en dicha arquitectura, particularmente *stdcall*.

El problema de la ISA x86 es que, en términos pedagógicos, es más difícil de estudiar a nivel práctico (se puede emular, pero el programa es viejo y no funciona en todos los sistemas operativos).

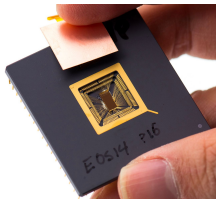
Por ese motivo, estudiaremos una ISA más actual: RISC-V.

Arquitectura RISC-V (RISC "*five*")

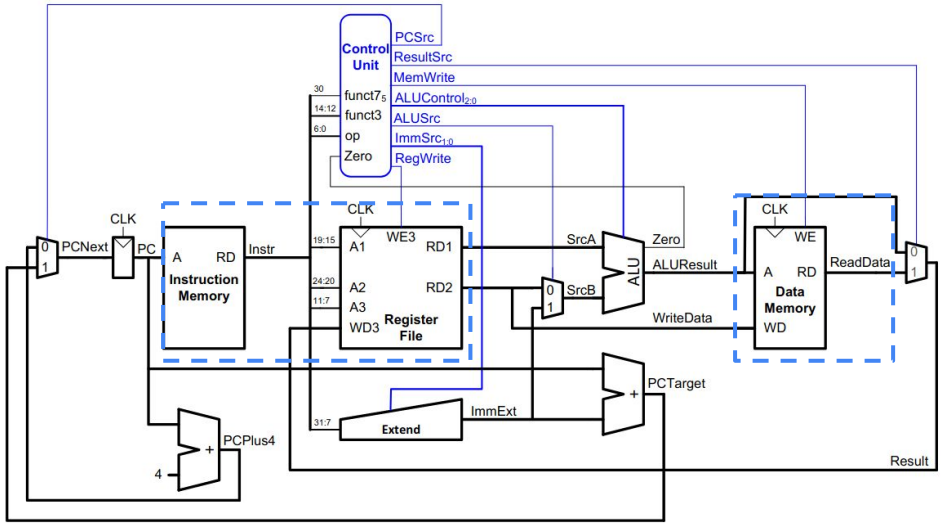
- ISA de tipo **RISC** *open source*. **No es una microarquitectura.**
- Propuesta por la Universidad de Berkeley el 2010 (pero con muchos colaboradores externos a la fecha).
- Arquitectura de tipo *load-store*. Posee dos categorías de instrucción: acceso de memoria y ALU (solo entre registros).
- Posee un diseño modular, *i.e.* que posee una ISA base y un conjunto de extensiones **opcionales**.

Arquitectura RISC-V - Ejemplo de diagrama reducido a un ciclo

- ***Register File*** posee el banco de 16 o 32 registros de la arquitectura.
- Presenta memoria de instrucciones y datos (Harvard).



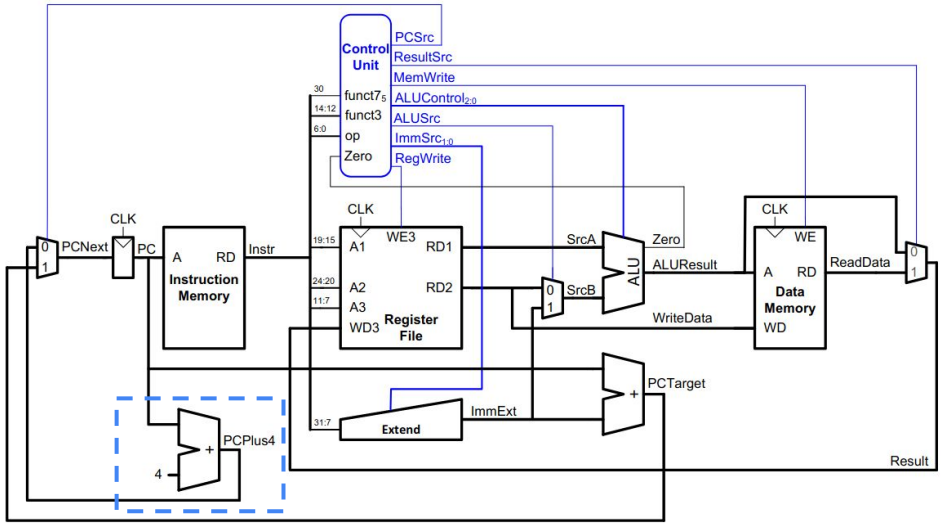
Primer prototipo (2013).



Este diagrama corresponde a una propuesta que permite implementar las instrucciones de la ISA RISC-V con ejecución en un ciclo.

Arquitectura RISC-V - Ejemplo de diagrama reducido a un ciclo

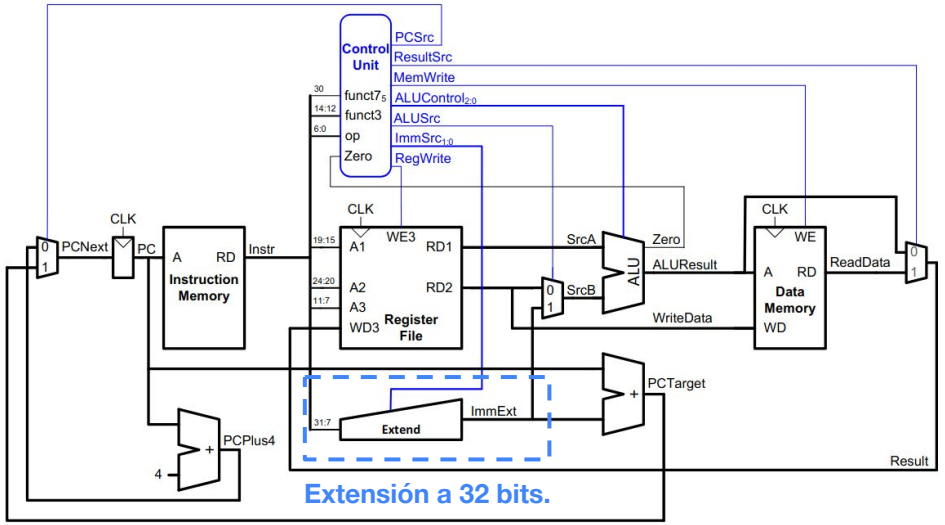
- Palabras de memoria de 8 bits (1 byte).
- Direcciones de memoria de 32 bits (4 bytes).
- Instrucciones de 32 bits (4 bytes). Ocupan 4 direcciones de memoria.



Como las instrucciones ocupan 4 direcciones de la memoria, PC + 4 apunta a la siguiente instrucción.

Arquitectura RISC-V - Ejemplo de diagrama reducido a un ciclo

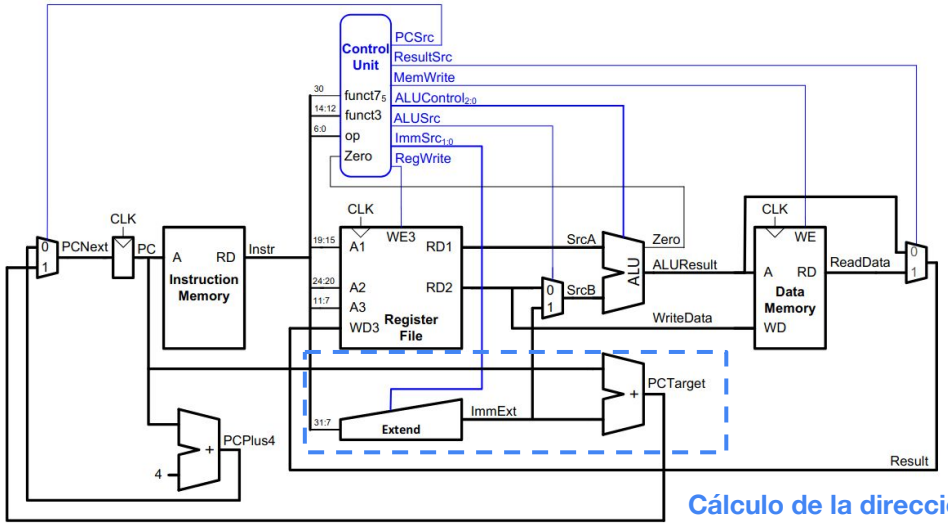
- Manejo de literales (llamados *immediates*).
- Como el literal ocupa menos de 32 bits dentro de la instrucción, "***Extend***" extiende su tamaño a 32 bits para operar con los valores de los registros. Esta extensión se realiza sobre el **bit de signo**.



Adicionalmente, el componente Extend recibe un *input* ImmSrc de 2 bits ya que, según el tipo de instrucción, pueden existir hasta cuatro formas de distribuir los bits del literal en ellas.

Arquitectura RISC-V - Ejemplo de diagrama reducido a un ciclo

- Las instrucciones de salto, a diferencia de las vistas en el computador básico, cargan la dirección de la instrucción mediante un *offset* que se suma con el PC actual.

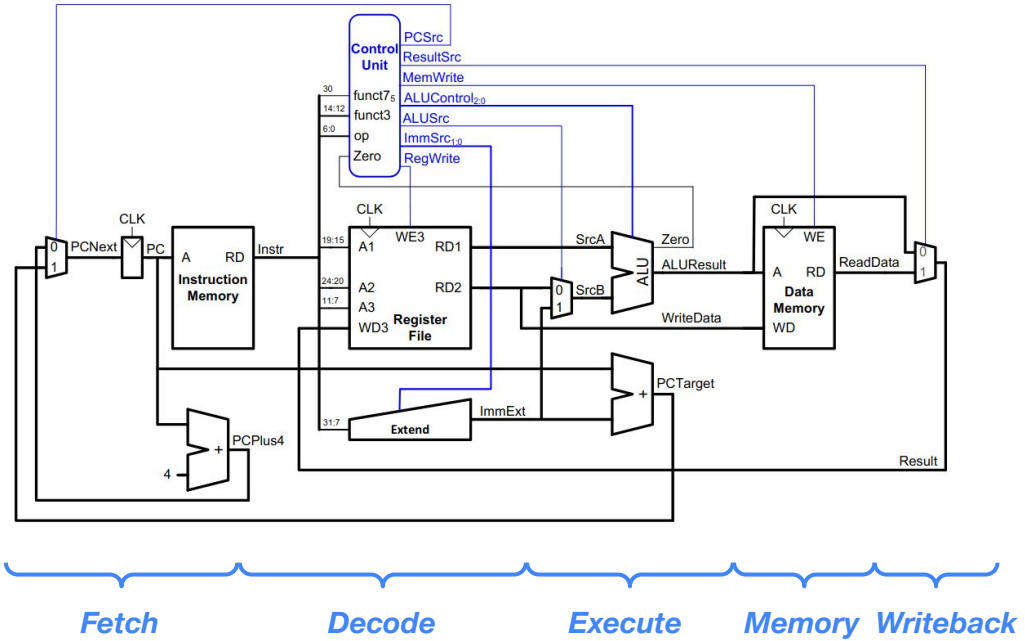


Cálculo de la dirección a saltar según el *offset* (que viene del literal proveniente de *Extend*).

Arquitectura RISC-V - Ejemplo de diagrama reducido a un ciclo

Se separa en 5 etapas:

- ***Fetch***: Obtención de la instrucción.
- ***Decode***: Obtención de señales de control.
- ***Execute***: Ejecución (ALU).
- ***Memory***: Lectura o escritura en memoria.
- ***Writeback***: escritura sobre el banco de registros.

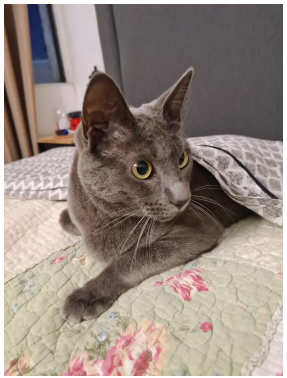


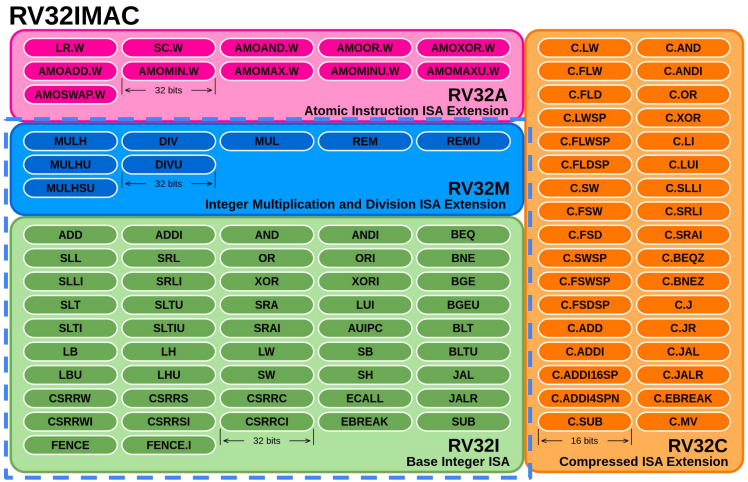
A pesar de la separación en etapas, las instrucciones se ejecutan en un ciclo. Más adelante veremos cómo paralelizar la ejecución de más de una instrucción en nuestra propia arquitectura.

Arquitectura RISC-V

- Se considera una arquitectura **Harvard + RISC**.

- Usaremos el conjunto base de instrucciones para números enteros (RV32I) y la extensión de multiplicación y división (RV32M).





Conjunto de instrucciones base y extensiones. En nuestro caso, usaremos la ISA RV32IM: RISC-V de 32 bits con extensiones *Integer* e *Integer Multiplication and Division*.

Arquitectura RISC-V - Microarquitectura en detalle

- 32 registros de propósito general de 32 bits dentro de un ***Register File***.
- La arquitectura varía según los módulos usados. RV32I posee una sola unidad de ejecución (ALU).
- Direcciones de memoria: 32 bits; palabras de memoria: 8 bits. Almacenamiento ***little endian***.
- *Stack* en memoria. Registro `SP` apunta al **último elemento ingresado al *stack***.

Arquitectura RISC-V - ISA en detalle

La ISA de la arquitectura RISC-V es un poco más compleja que la del computador básico (a pesar de ser RISC):

- Posee instrucciones de transferencia, aritméticas, lógicas, saltos y subrutinas.
- Posee tipos de instrucción. Cada tipo posee un formato distinto (a diferencia del computador básico donde se tiene solo *opcode* y literal concatenados para toda instrucción).
- Acepta tipos de datos nativos de 8 y 32 bits (con y sin signo).
- Posee un solo tipo de direccionamiento: **indirecto + *offset***.

Arquitectura RISC-V - ISA en detalle

Listado de registros

Registro(s)	Mnemotecnia ABI	Descripción
`x0`	`zero`	Registro cero. Almacena este valor y **no cambia**. Ignora las escrituras.
`x1`	`ra`	***R**eturn **A**ddress*. Almacena la dirección de retorno de las subrutinas.
`x2`	`sp`	***S**tack **P**ointer*, apunta al último elemento almacenado.
`x3`	`gp`	***G**lobal **P**ointer*, apunta al segmento de memoria donde se almacenan las variables globales.
`x4`	`tp`	***T**hread **P**ointer*, apunta al segmento de memoria donde se almacenan las variables de un *thread* para aplicaciones de múltiples *threads*.
`x5-x7, x28-x31`	`t0-t6`	Registros temporales. Pierden su valor entre llamados de subrutinas.
`x8-x9, x18-x27`	`s0-s11`	Registros guardados (*saved*). Preservan su valor entre llamados de subrutinas.
`x10-x17`	`a0-a7`	Registros para argumentos de subrutinas.
`x10-x11`	`a0-a1`	Si bien son de argumentos de subrutinas, también se utilizan para almacenar valores de retorno.

A nivel de Assembly, usaremos los nombres mnemotécnicos para referirnos a estos registros, respetando la ABI (Application Binary Interface).

Si bien todos son de propósito general, por convención los usaremos según las descripciones de este listado (más información en las siguientes diapositivas).

Arquitectura RISC-V - ISA en detalle

Listado de directivas de Assembler

Directiva de Assembler	Descripción
`.text`	**Segmento de texto (código).**
`.data`	**Sección de datos global.**
`.bss`	Sección de datos globales inicializados en 0.
`.section .foo`	Sección llamada "`foo`".
`.align N`	Alinear el siguiente dato/instrucción a 2^N bytes de memoria.
`.balign N`	Alinear el siguiente dato/instrucción a *N* bytes de memoria.
`.globl sym`	***Label*** `sym` **se vuelve global.**
`.string "str"`	Almacena el *string* "`str`" en memoria.
`.word w1, w2, ..., wN`	**Almacena *N* valores de 32 bits en palabras de memoria sucesivas.**
`.byte w1, w2, ..., wN`	**Almacena *N* valores de 8 bits en bytes de memoria sucesivos.**
`.space N`	Reserva *N* bytes para almacenar una variable.
`.equ name, constant`	Define el símbolo `name` con valor constant.
`.end`	Término del código Assembly.

Las directivas de un Assembler consisten en órdenes para que este tome ciertas acciones o realice cambios de configuración.

¡No son instrucciones ni se traducen a código de máquina!

*** `.align`, `.balign` agregan *padding* de bytes "basura" en memoria de forma que la inserción de la siguiente variable quede almacenada en una dirección "alineada" (generalmente una dirección par para optimizar direccionamiento).**

Arquitectura RISC-V - ISA en detalle

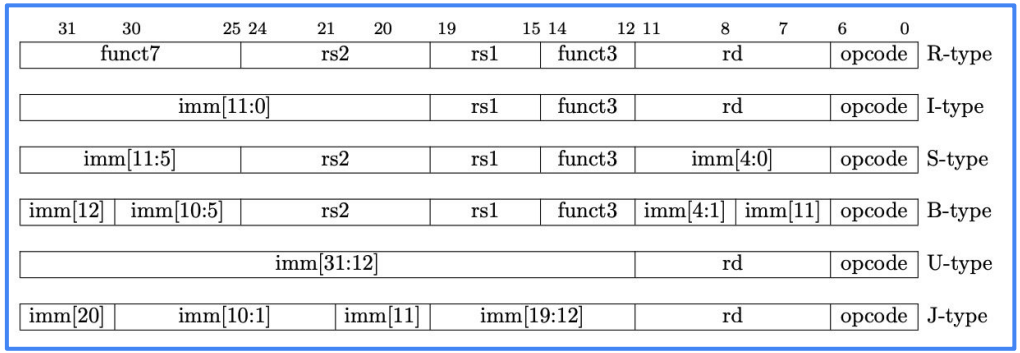
Tipos de instrucción

- I-Type: ***I****mmediate Type*. Utilizan al menos un literal de operando.
- R-Type: ***R****egister Type*. Utilizan solo registros como operandos.
- S-Type: ***S****tore Type*. Almacenamiento en la memoria de datos.
- B-Type: ***B****ranching Type*, variante de S-Type. Saltos condicionales a través de *offsets*.
- U-Type: ***U****pper Immediate Type*. Permite cargar en registros literales de más de 12 bits (límite de instrucciones I-Type).
- J-Type: ***J****ump Type*, variante de U-Type. Saltos incondicionales para las subrutinas.

Arquitectura RISC-V - ISA en detalle

Formato de tipos de instrucción

- Instrucciones de 32 bits.
- `imm` = literal (*immediate*).
- `rs1, rs2` = registros de operandos.
- `rd` = registro de destino.
- *opcode* = identificador de la instrucción.
- `funct7, funct3` = Bits adicionales que, en conjunto con el *opcode*, definen la operación completa a ejecutar.



Arquitectura RISC-V - ISA en detalle

Resumen de instrucciones - Operaciones aritméticas

Mnemotecnia	Instrucción	Tipo	Descripción
`ADD rd, rs1, rs2`	**Adición**	**R**	`rd ← rs1 + rs2`
`SUB rd, rs1, rs2`	**Sustracción**	**R**	`rd ← rs1 - rs2`
`ADDI rd, rs1, imm12`	**Adición de literal**	**I**	`rd ← rs1 + imm12`
`SLT rd, rs1, rs2`	Configurar “menor a”	R	`rd ← rs1 < rs2 ? 1 : 0`
`SLTI rd, rs1, rs2`	Configurar “menor a” literal	I	`rd ← rs1 < imm12 ? 1 : 0`
`SLTU rd, rs1, rs2`	Configurar “menor a” sin signo	R	`rd ← rs1 < rs2 ? 1 : 0`
`SLTIU rd, rs1, imm12`	Configurar “menor a” literal sin signo	I	`rd ← rs1 < imm12 ? 1 : 0`
`LUI rd, imm20`	Cargar literal “superior” (20 bits)	U	`rd ← imm20 << 12 (SHL 12)`
`AUIP rd, imm20`	Sumar literal “superior” a PC (20 bits)	U	`rd ← PC + imm20 << 12 (SHL 12)`

* Para restar un literal, usamos `ADDI` con un literal negativo.

Arquitectura RISC-V - ISA en detalle

Resumen de instrucciones - Operaciones lógicas 1/2

Mnemotecnia	Instrucción	Tipo	Descripción
`AND rd, rs1, rs2`	Operación AND	R	`rd ← rs1 & rs2`
`OR rd, rs1, rs2`	Operación OR	R	`rd ← rs1 | rs2`
`XOR rd, rs1, rs2`	Operación XOR	R	`rd ← rs1 ^ rs2`
`ANDI rd, rs1, imm12`	Operación AND con literal	I	`rd ← rs1 & imm12`
`ORI rd, rs1, imm12`	Operación OR con literal	I	`rd ← rs1 | imm12`
`XORI rd, rs1, imm12`	Operación XOR con literal	I	`rd ← rs1 ^ imm12`

Arquitectura RISC-V - ISA en detalle

Resumen de instrucciones - Operaciones lógicas 2/2

Mnemotecnia	Instrucción	Tipo	Descripción
`SLL rd, rs1, rs2`	Operación *shift left* lógico	R	`rd ← rs1 << rs2`
`SRL rd, rs1, rs2`	Operación *shift right* lógico	R	`rd ← rs1 >> rs2`
`SRA rd, rs1, rs2`	Operación *shift right* aritmético	R	`rd ← rs1 >> rs2`
`SLLI rd, rs1, shamt`	Operación *shift left* lógico con literal	I	`rd ← rs1 << shamt`
`SRLI rd, rs1, shamt`	Operación *shift right* lógico con literal	I	`rd ← rs1 >> shamt`
`SRAI rd, rs1, shamt`	Operación *shift right* aritmético con literal	I	`rd ← rs1 >> shamt`

* `shamt` o *shift amount* es la cantidad de *shifts* a realizar y se codifica como un entero a partir de los 5 bits menos significativos del literal (`imm12[4:0]`).

Arquitectura RISC-V - ISA en detalle

Resumen de instrucciones - Operaciones de carga y almacenamiento

Mnemotecnia	Instrucción	Tipo	Descripción
`LW rd, imm12(rs1)`	**Cargar *word* (32 bits)**	**I**	**`rd ← mem[rs1 + imm12]`**
`LH rd, imm12(rs1)`	Cargar *half word* (16 bits)	I	`rd ← mem[rs1 + imm12]`
`LB rd, imm12(rs1)`	Cargar byte (8 bits)	I	`rd ← mem[rs1 + imm12]`
`LWU rd, imm12(rs1)`	Cargar *word* sin signo (32 bits)	I	`rd ← mem[rs1 + imm12]`
`LHU rd, imm12(rs1)`	Cargar *half word* sin signo (16 bits)	I	`rd ← mem[rs1 + imm12]`
`LBU rd, imm12(rs1)`	Cargar byte sin signo (8 bits)	I	`rd ← mem[rs1 + imm12]`
`SW rs2, imm12(rs1)`	**Almacenar *word* (32 bits)**	**S**	**`rs2 → mem[rs1 + imm12]`**
`SH rs2, imm12(rs1)`	Almacenar *half word* (16 bits)	S	`rs2(15:0) → mem[rs1 + imm12]`
`SB rs2, imm12(rs1)`	Almacenar byte (8 bits)	S	`rs2(7:0) → mem[rs1 + imm12]`

* Para direccionar en `LW` y `SW`, se usa el formato `offset(x)`, donde la dirección se almacena en el registro `x`. Ejemplo: `4(sp)`

Arquitectura RISC-V - ISA en detalle

Resumen de instrucciones - Saltos y subrutinas

Mnemotecnia	Instrucción	Tipo	Descripción
`BEQ rs1, rs2, imm12`	**Salto con condición “igual”**	**B**	`if rs1 == rs2: PC ← PC + imm12`
`BNE rs1, rs2, imm12`	**Salto con condición “distinto”**	**B**	`if rs1 != rs2: PC ← PC + imm12`
`BGE rs1, rs2, imm12`	**Salto con condición “mayor o igual”**	**B**	`if rs1 >= rs2: PC ← PC + imm12`
`BGEU rs1, rs2, imm12`	Salto con condición “mayor o igual” sin signo	B	`if rs1 >= rs2: PC ← PC + imm12`
`BLT rs1, rs2, imm12`	**Salto con condición “menor”**	**B**	`if rs1 < rs2: PC ← PC + imm12`
`BLTU rs1, rs2, imm12`	Salto con condición “menor” sin signo	B	`if rs1 < rs2: PC ← PC + imm12`
`JAL rd, imm20`	**Salto incondicional con “enlace”**	**J**	`rd ← PC+4; PC ← PC + imm20`
`JALR rd, imm12(rs1)`	**Salto incondicional con “enlace” a registro**	**I**	`rd ← PC+4; PC ← rs1 + imm12`

* En estos casos, el literal representa el *offset* para llegar a la instrucción deseada desde el *Program Counter*. A nivel de código, se observa como el *label* de la dirección a saltar.

Arquitectura RISC-V - ISA en detalle

Resumen de instrucciones - Pseudo-instrucciones 1/2

Mnemotecnia	Instrucción	Instrucción(es) base
`LI rd, imm12`	**Cargar literal en registro que utiliza ≤ 12 bits**	`ADDI rd, zero, imm12`
`LI rd, imm`	**Cargar literal en registro que utiliza > 12 bits**	`LUI rd, imm[31:12]; ADDI rd, rd, imm[11:0]`
`LA rd, sym`	**Cargar dirección en registro**	`AUIPC rd, sym[31:12]; ADDI rd, rd, sym[11:0]`
`MV rd, rs`	**Copiar registro**	`ADDI rd, rs, 0`
`NOT rd, rs`	**Complemento de 1**	`XORI rd, rs, -1`
`NEG rd, rs`	**Complemento de 2**	`SUB rd, zero, rs`
`BGT rs1, rs2, offset`	**Salto si `rs1 > rs2`**	`BLT rs2, rs1, offset`
`BLE rs1, rs2, offset`	**Salto si `rs1 ≤ rs2`**	`BGE rs2, rs1, offset`
`BGTU rs1, rs2, offset`	Salto si `rs1 > rs2` sin signo	`BLTU rs2, rs1, offset`
`BLEU rs1, rs2, offset`	Salto si `rs1 ≤ rs2` sin signo	`BGEU rs2, rs1, offset`

* Las pseudo-instrucciones mnemotécnicas se traducen a las instrucciones reales de RISC-V. Esto ayuda a tener instrucciones de operaciones útiles en un lenguaje más sencillo de entender.

Arquitectura RISC-V - ISA en detalle

Resumen de instrucciones - Pseudo-instrucciones 2/2

Mnemotecnia	Instrucción	Instrucción(es) base
`BEQZ rs1, offset`	**Salto si `rs1 = 0`**	`BEQ rs1, zero, offset`
`BNEZ rs1, offset`	**Salto si `rs1 ≠ 0`**	`BNE rs1, zero, offset`
`BGEZ rs1, offset`	**Salto si `rs1 ≥ 0`**	`BGE rs1, zero, offset`
`BLEZ rs1, offset`	**Salto si `rs1 ≤ 0`**	`BGE zero, rs1, offset`
`BGTZ rs1, offset`	**Salto si `rs1 > 0`**	`BLT zero, rs1, offset`
`J offset`	**Salto incondicional**	`JAL zero, offset`
`CALL offset12`	**Llamado a subrutina (dirección ≤ 12 bits)**	`JALR ra, ra, offset12`
`CALL offset`	**Llamado a subrutina (dirección > 12 bits)**	`AUIPC ra, offset[31:12]; JALR ra, ra, offset[11:0]`
`RET`	**Retorno de la subrutina**	`JALR zero, 0(ra)`
`NOP`	No se realiza ninguna operación	`ADDI zero, zero, 0`

Arquitectura RISC-V - ISA en detalle

Ejemplo de código - Multiplicación

```
a = 10
b = 200
res = 0
while (a > 0):
    res += b
    a -= 1
print(res)
```

Programa en pseudocódigo (Python).

```
.globl main
.text
  main:
    li t0, 10                   # t0 = 10
    li t1, 200                  # t1 = 200
    li t2, 0                    # t2 = 0
    mul_loop:
      beq t0, zero, end         # if t0 = 0 jmp end
      add t2, t2, t1            # t2 += t1
      addi t0, t0, -1           # t0 += -1
      j mul_loop                # jal zero, mul_loop
    end:
.end
```

Programa en RISC-V. Notar el uso de pseudo-instrucciones para facilitar la lectura del código.

Arquitectura RISC-V - ISA en detalle

Resumen de instrucciones - Extensión M

Mnemotecnia	Instrucción	Tipo	Descripción
`MUL rd, rs1, rs2`	**32 bits menos significativos del producto.**	**R**	**`rd ← (rs1 * rs2)[31:0]`**
`MULH rd, rs1, rs2`	32 bits más significativos del producto (`rs1`, `rs2` con signo).	R	`rd ← (rs1 * rs2)[63:32]`
`MULHSU rd, rs1, rs2`	32 bits más significativos del producto (`rs1` con signo, `rs2` sin signo).	R	`rd ← (rs1 * rs2)[63:32]`
`MULHU rd, rs1, rs2`	32 bits más significativos del producto (`rs1`, `rs2` sin signo).	R	`rd ← (rs1 * rs2)[63:32]`
`DIV rd, rs1, rs2`	**División con signo**	**R**	**`rd ← rs1 / rs2`**
`DIVU rd, rs1, rs2`	División sin signo	R	`rd ← rs1 / rs2`
`REM rd, rs1, rs2`	Resto con signo	R	`rd ← rs1 % rs2`
`REMU rd, rs1, rs2`	Resto sin signo	R	`rd ← rs1 % rs2`

Arquitectura RISC-V - ISA en detalle

Ejemplo de código - Multiplicación con extensión M

```
.globl main
.text
  main:
    li t0, 10                   # t0 = 10
    li t1, 200                  # t1 = 200
    mul t2, t0, t1              # t2 = t0 * t1 = 2000
.end
```

Si agregamos la extensión RV31M al conjunto de instrucciones base RV31I, entonces el código se simplifica de manera significativa (similar a lo que ocurre con las operaciones MUL y DIV de la arquitectura x86).

Arquitectura RISC-V - ISA en detalle

Instrucción ECALL (*Environment Call*)

- Instrucción especial de tipo I que permite realizar una solicitud al entorno de ejecución (sistema operativo).

- Su uso varía según la implementación, pero en este curso nos basaremos en la implementación del simulador de RISC-V RARS (*RISC-V Assembler And Runtime System*).

Arquitectura RISC-V - ISA en detalle

Instrucción ECALL (*Environment Call*)

- Se hace uso de un código que indica la llamada a efectuar. Este se almacena en el registro `a7` y los argumentos en `a0-a6`.

- Existen múltiples funciones, pero usaremos dos:
  - `PrintInt(code=1)`: Impresión de valor entero en consola. Solo recibe un argumento en `a0`, el número a imprimir.
  - `Exit(code=10)`: Término de programa. No recibe argumentos y asume que el código de término es 0 (sin errores).

Arquitectura RISC-V - ISA en detalle

Ejemplo de código - Impresión de número y salida del programa

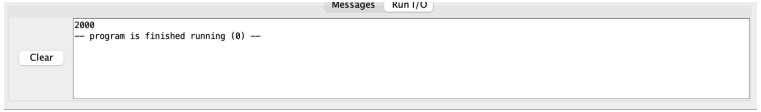
```
.globl main
.text
  main:
    li a0, 11                   # a0 = 11
    li a7, 1                    # a7 = 1 (PrintInt)
    ecall                       # Imprime 11 en consola
    li a7, 10                   # a7 = 10 (Exit)
    ecall                       # Termina el programa
```

Ejemplo de programa que imprime en consola el número 11. Se incluye además el resultado del simulador RARS. Cabe destacar que RARS no reconoce la directiva `.end`, por lo que es necesario usar `ecall` para el término del programa.

Arquitectura RISC-V - ISA en detalle

Ejemplo de código - Multiplicación con extensión M en RARS

```
.globl main
.text
  main:
    li t0, 10                 # t0 = 10
    li t1, 200                # t1 = 200
    mul t2, t0, t1            # t2 = t0 * t1 = 2000
    mv a0,t2                  # a0 = t2
    li a7,1                   # a7 = 1 (PrintInt)
    ecall                     # Print a0 = 2000
    li a7,10                  # a7 = 10 (Exit)
    ecall                     # Exit
```



Ejemplo de código de multiplicación ejecutable en RARS.

Arquitectura RISC-V - Convención de llamada

Al igual que con la arquitectura x86, se hace uso de convenciones de llamada para estandarizar la forma en la que se ejecutan las subrutinas en un programa.

En el caso de RISC-V, se hace uso de una convención propia donde el almacenamiento de parámetros, valores de retorno y dirección de retorno **recaen en los registros** y no en el *stack*. Este último se utiliza como respaldo para **preservar el valor de los registros entre llamadas**.

Arquitectura RISC-V - Convención de llamada

- El registro `ra` (`x1`) almacena la dirección de retorno.
- Los argumentos de una subrutina se almacenan en los registros `a0-a7` (`x10-x17`).
- Si existen valores de retorno, se almacenan en los registros `a0-a1` (`x10-x11`).
- Se hace uso del *stack* con el registro `sp` (`x2`). Se puede utilizar para incluir argumentos adicionales, variables locales y **respaldo de registros**.

Arquitectura RISC-V - Convención de llamada

El encargado de respaldar el valor de los registros con el llamado de una subrutina varía según el tipo de registro. En este caso, el encargado puede ser quien llama a la subrutina (*caller*) o la subrutina llamada (*callee*).

El respaldo de los registros se realiza a través del *stack pointer*, modificando su valor para reservar valores en la memoria de *stack*.

Registro(s)	Mnemotecnia ABI	Encargado del respaldo
x0	zero	-
x1	ra	*Caller*
x2	sp	*Callee*
x3	gp	-
x4	tp	-
x5-x7, x28-x31	t0-t6	*Caller*
x8-x9, x18-x27	s0-s11	*Callee*
x10-x17	a0-a7	*Caller*
x10-x11	a0-a1	*Caller*

Arquitectura RISC-V - Convención de llamada

Ejemplo de código - Duplicación de elementos de un arreglo

```
.data
    len: .word 5
    arr: .word 198, 137, 42, 63, 175
.text
    start:
        li s0, 4                        # s0 = 4 bytes por dirección
        lw s1, len                      # s1 = largo del arreglo
        li t0, 0                        # Contador (i)
        la t1, arr                      # t1 = dirección del arreglo (inicialmente arr[0])
        mul t2, s0, s1                  # t2 = s0 * s1 = bytes que ocupa el arreglo de entrada
        add t2, t2, t1                  # t2 += t1 = primera dirección que podemos usar de .data
        while:
            lw a0, 0(t1)                # a0 = arr[i]
Preámbulo   addi sp, sp, -12
            sw t0, 0(sp)                # Respaldamos t0, t1 y ra (caller-saved). t1 se respalda
            sw t1, 4(sp)                # aunque no se use porque call lo modifica con la
            sw ra, 8(sp)                # dirección de retorno en RARS
            call double_give_next_person
Prólogo     lw t0, 0(sp)                # Recuperamos t0, t1 y ra y restauramos el stack
            lw t1, 4(sp)
            lw ra, 8(sp)
            addi sp, sp, 12
            sw a0, 0(t2)                # out[i] = a0
            addi t0, t0, 1              # t0 += 1
            beq t0, s1, end             # Termina cuando se recorre todo el arreglo
            add t1, t1, s0              # t1 += s0 = dirección arr[i+1]
            add t2, t2, s0              # t2 += s0 = dirección out[i+1]
            j while
    double_give_next_person:
        mv t0, a0
        add a0, a0, t0                  # a0 = a0 + t0 = 2 * a0
        ret
    end:
        li a7, 10
        ecall
```

Programa en RISC-V que duplica los elementos de un arreglo y los almacena en las direcciones posteriores a dicha variable. El respaldo de registros *caller-saved* en la convención de llamada se realiza de la siguiente forma:

1. Los registros se respaldan en el *stack* antes de la llamada. Para ello, se desplaza el registro `sp` según la cantidad de bytes a respaldar (4 bytes por registro) y luego se almacenan a través de direccionamiento indirecto por registro `sp` con *offset*. Esto se conoce como el **preámbulo**.
2. Finalizada la llamada, se restauran los registros desde el *stack* y se restablece el valor de `sp`. Esto se conoce como el **prólogo**.

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	5	198	137	42	63	175	396	274
0x10010020	84	126	350	0	0	0	0	0

Resultado de la ejecución en el segmento de datos en RARS.

Arquitectura RISC-V - Convención de llamada

Ejemplo de código - Duplicación de elementos de un arreglo

```
.data
    len: .word 5
    arr: .word 198, 137, 42, 63, 175
.text
    start:
        li s0, 4                        # s0 = 4 bytes por dirección
        lw s1, len                      # s1 = largo del arreglo
        li t0, 0                        # Contador (i)
        la t1, arr                      # t1 = dirección del arreglo (inicialmente arr[0])
        mul t2, s0, s1                  # t2 = s0 * s1 = bytes que ocupa el arreglo de entrada
        add t2, t2, t1                  # t2 += t1 = primera dirección que podemos usar de .data
        while:
            lw a0, 0(t1)                # a0 = arr[i]
            addi sp, sp, -12
            sw t0, 0(sp)                # Respaldamos t0, t1 y ra (caller-saved). t1 se respalda
            sw t1, 4(sp)                # aunque no se use porque call lo modifica con la
            sw ra, 8(sp)                # dirección de retorno en RARS
            call double_give_next_person
            lw t0, 0(sp)                # Recuperamos t0, t1 y ra y restauramos el stack
            lw t1, 4(sp)
            lw ra, 8(sp)
            addi sp, sp, 12
            sw a0, 0(t2)                # out[i] = a0
            addi t0, t0, 1              # t0 += 1
            beq t0, s1, end             # Termina cuando se recorre todo el arreglo
            add t1, t1, s0              # t1 += s0 = dirección arr[i+1]
            add t2, t2, s0              # t2 += s0 = dirección out[i+1]
            j while
    double_give_next_person:
        mv t0, a0
        add a0, a0, t0                  # a0 = a0 + t0 = 2 * a0
        ret
    end:
        li a7, 10
        ecall
```

En este mismo ejemplo, es importante destacar que el registro `ra` podría no respaldarse y que se asegure el funcionamiento correcto del programa. No obstante, por convención es buena práctica hacerlo de todas formas ya que el programa podría ser ejecutado mediante una llamada desde otro archivo. Si no se respaldara `ra`, **se perdería la dirección de retorno para el programa que ejecuta este ejemplo**. El no respaldo de `ra` generará problemas **siempre que haya llamados de subrutinas anidados**.

Por otra parte, `t1` se respalda solo porque la instrucción `call` de RARS modifica su valor con la dirección de retorno, en estricto rigor el respaldo no es necesario si no se utiliza dentro de la subrutina.

Arquitectura RISC-V - Convención de llamada

Ejemplo de código - Duplicación de elementos de un arreglo

```
.data
    len: .word 5
    arr: .word 198, 137, 42, 63, 175
.text
    start:
        li t0, 4                        # s0 = 4 bytes por dirección
        lw t2, len                      # s1 = largo del arreglo
        li s0, 0                        # Contador (i)
        la s1, arr                      # t1 = dirección del arreglo (inicialmente arr[0])
        mul s2, t0, t2                  # t2 = s0 * s1 = bytes que ocupa el arreglo de entrada
        add s2, s2, s1                  # t2 += t1 = primera dirección que podemos usar de .data
        while:
            lw a0, 0(s1)                # a0 = arr[i]
Preámbulo { addi sp, sp, -4
            sw ra, 0(sp)                # Respaldamos ra (caller-saved)
            call double_give_next_person
            lw ra, 0(sp)                # Recuperamos ra y restauramos el stack
Prólogo   { addi sp, sp, 4
            sw a0, 0(s2)                # out[i] = a0
            addi s0, s0, 1              # s0 += 1
            beq s0, t2, end             # Termina cuando se recorre todo el arreglo
            add s1, s1, t0              # s1 += t0 = dirección arr[i+1]
            add s2, s2, t0              # s2 += t0 = dirección out[i+1]
            j while
    double_give_next_person:
Preámbulo { addi sp, sp, -4
            sw s0, 0(sp)            # Respaldamos s0 (callee-saved)
            mv s0, a0
Prólogo   { add a0, a0, s0          # a0 = a0 + s0 = 2 * a0
            lw s0, 0(sp)            # Recuperamos s0 y restauramos el stack
            addi sp, sp, 4
            ret
    end:
        li a7, 10
        ecall
```

Veamos el mismo ejemplo pero invirtiendo los registros `s*` por registros `t*`. El respaldo de registros *callee-saved* en la convención de llamada se realiza de la siguiente forma:

1. Los registros se respaldan en el *stack* al comienzo de la llamada al igual que con los registros *caller-saved*. Esto también es un **preámbulo**.
2. Antes de finalizar la llamada con `ret`, se restauran los registros desde el *stack* y se restablece el valor de `sp` al igual que con los registros *caller-saved*. Esto también es un **prólogo**.

En este caso seguimos respaldando el registro `ra` por ser *caller-saved* y por lo comentado en la diapositiva anterior.

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	5	198	137	42	63	175	396	274
0x10010020	84	126	350	0	0	0	0	0

Resultado de la ejecución en el segmento de datos en RARS.

Arquitectura RISC-V - Convención de llamada

Ejemplo de código - Factorial, función recursiva

```
.data
    N: .word 4                  # N = Argumento de factorial. Calcularemos N! = 4
.text
    main:
        addi sp, sp, -4         # Reservamos 4 bytes en el stack
        sw   ra, 0(sp)          # Respaldamos ra
        la   t0, N              # Dirección de memoria de N
        lw   t0, 0(t0)          # Valor de N
        add  a0, zero, t0       # Argumento 0 = valor de N
        call factorial          # factorial(N)
        li   a7, 1              # Llamada de sistema: print int
        ecall                   # Valor en consola: 24 (a0, valor de retorno)
        lw   ra, 0(sp)          # Restauramos ra
        addi sp, sp, 4          # Restauramos el stack
        li   a7, 10             # Llamada de sistema: exit
        ecall
    factorial:
        addi sp, sp, -8         # Reservamos 8 bytes en el stack
        sw   ra, 0(sp)          # Respaldamos ra
        sw   a0, 4(sp)          # Respaldamos N
        blez a0, factorial_zero # if (N > 0){
        addi a0, a0, -1         #   N -= 1
        call factorial          #   (N-1)! = factorial(N-1)
        lw   t0, 4(sp)          #   Recuperamos N
        mul  a0, a0, t0         #   N! = N * (N-1)! = N * factorial(N-1)
        j    factorial_end      # }
        factorial_zero:         # else {
            li   a0, 1          #   N! = 1
        factorial_end:          # }
            lw   ra, 0(sp)      # Restauramos solo ra, a0 ahora posee el retorno
            addi sp, sp, 8      # Restauramos el stack
        ret
```

Programa en RISC-V que calcula el valor del factorial de la variable *N*.

Ahora, **el *callee* también actúa como *caller* por la recursión** y respalda tanto el registro `ra` como el registro de argumento `a0`. Además, el callee se encarga de restaurar el registro `sp` en cada llamada, **lo que asegura la consistencia de su valor al realizar el último retorno**. En este caso no se restaura `a0` ya que este posee el valor de retorno a ser utilizado en la impresión de la consola.

Si se hubieran utilizado registros `s*`, también habrían sido respaldados por el *callee* como en el ejemplo anterior.

Arquitectura RISC-V - Resumen de la convención de llamada

- Si usamos los registros `t*` tanto fuera como dentro de una subrutina, los respaldamos en el *stack* antes del llamado y los restauramos posterior a este.
- Si usamos los registros `s*` tanto fuera como dentro de una subrutina, los respaldamos en el *stack* al principio de la subrutina y los recuperamos justo antes del retorno.
- Si usamos los registros `a*`, los respaldamos dentro de la subrutina pero los recuperamos siempre que no interfieran con el valor de retorno (`a2-a7`).
- Por seguridad, siempre respaldamos el registro ra antes de un llamado.

Instalación de RARS

En vez de hacer ejercicios, realizaremos la instalación de RARS para poder ejecutar código RISC-V en nuestros computadores.

Para ello, es necesario instalar Java 8 o superior y seguir las instrucciones del repositorio principal.

Instalación de RARS - Instalación de Java 8

- **Windows:** Descargar Java 8 en el siguiente enlace y ejecutar el archivo `.jar`.

- **Mac**
  - **Chip M1:** Seguir el siguiente tutorial.
  - **Modelos previos:** Descargar Java 8 en el siguiente enlace y ejecutar el archivo .jar (con click derecho por seguridad).

- **Ubuntu:** Seguir el siguiente tutorial.

* Pueden instalar versiones superiores de Java, la versión 8 es la mínima requerida.

Tarea 3 - Revisión Pregunta 1

Para finalizar la clase, revisaremos las preguntas de RISC-V de la tarea 3 y resolveremos un problema que nos ayudará a avanzar en ella.

Tarea 3 - Revisión Pregunta 1

Pregunta 1.a.

(a) (4 ptos.) La especificación de RISC-V incluye una convención de llamada, que a partir de la versión V20191213 se encuentra definida como parte del psABI. Investigue y escriba brevemente qué indica la convención de llamada para la extensión base RV32I, los aspectos más relevantes de esta, cómo se utiliza y por qué es necesaria. Para las siguientes secciones de la tarea deberá escribir programas en *assembly* RISC-V, por lo que es importante que preste atención a cómo se utiliza la convención de llamada.

Tarea 3 - Revisión Pregunta 1

Pregunta 1.b.

(b) (4 ptos.) Para esta pregunta, deberá desarrollar un programa que detecte si un número es primo gemelo. Un número primo p es gemelo si existe otro número primo q tal que $|p-q| = 2$. En este programa, recibirá en la sección `.data` una etiqueta `N`, que corresponde a un entero positivo. Al finalizar la ejecución, deberá usar el `ecall 1` de RARS para imprimir en consola un 1 si N es primo gemelo y un 0 en otro caso. La primera parte de su archivo se debiese ver de la siguiente forma:

```
.globl start
.data
    N: .word 11
.text
    start:
        # do stuff
```

Se evaluará la correctitud del código, que este respete la convención de llamada y que pueda aceptar un número N arbitrario como *input*.

Tarea 3 - Revisión Pregunta 1

Pregunta 1.c.

(c) (8 ptos.) Para esta pregunta, deberá programar una pequeña versión de juego FizzBuzz en RISC-V ASM. Para esto, recibirá en la sección `.data` una etiqueta `N`, que corresponde a un entero positivo y deberá escribir, a partir de la etiqueta `out`, todos los números entre 0 y `N` como `.word`, con la salvedad de que si el número es divisible por 3, este se reemplaza por 70 (ASCII "F"); en caso de ser divisible por 5, se reemplaza por 66 (ASCII "B"); y si es divisible por ambos, se reemplaza por 7066. Además, si un número es primo gemelo, deberá reemplazarlo por 80 (ASCII "P"). A modo de ejemplo, si $N = 15$ la secuencia que se guardará en `out` será: `7066, 1, 2, 80, 4, 80, 6, 80, 8, 70, 66, 80, 70, 80, 14, 7066`
La primera parte de su archivo se debiese ver de la siguiente forma:

```
.globl start
.data
    N: .word 10
    out: .word
.text
    start:
        # do stuff
```

Se evaluará la correctitud del código, que este respete la convención de llamada y que pueda aceptar un número N arbitrario como *input*.

Tarea 3 - Revisión Pregunta 1

Pregunta 1.d.

(d) (14 ptos.) Para esta pregunta, deberá resolver un puzzle utilizando RISC-V ASM. Recibirá como *input* un entero `N` y dos arreglos, `M` y `T`. `N` representa la cantidad de nodos de un grafo; `M` representa su matriz de adyacencia; y `T` representa el tipo de cada uno de sus nodos. Existen cuatro tipos de nodo:

- `Inicio` = `1`
- `Final` = `2`
- `Punto` = `3`
- `Vacío` = `0`

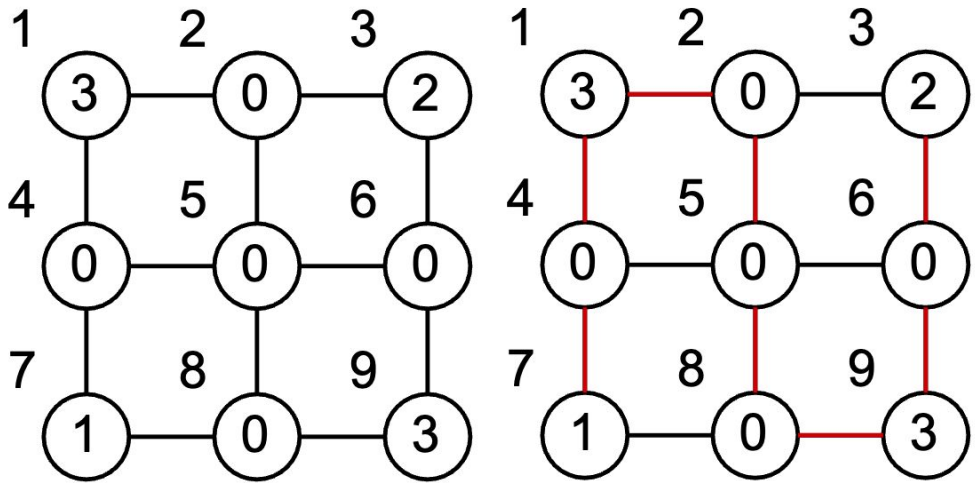
El objetivo del puzzle es trazar un camino desde `Inicio` hasta `Final`, conectado a todos los nodos `Punto`. El camino no puede pasar dos veces por el mismo vértice ni borde.

Como resultado, su programa deberá usar el `ecall 1` de RARS para imprimir, en orden, los vértices que recorre el camino de la solución.

Tarea 3 - Revisión Pregunta 1

Pregunta 1.d.

A modo de ejemplo, un posible puzzle junto con su matriz de adyacencia, lista de tipo y solución, podría ser el siguiente:



Puzzle

Solución

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Matriz de adyacencia

$$3, 0, 2, 0, 0, 0, 1, 0, 3$$

Lista de tipos de nodo

Tarea 3 - Revisión Pregunta 1

Pregunta 1.d.

El *input* asociado podría verse de la siguiente manera:

```
.globl start
.data
    # --- No modificar labels ---
    N: .word 9
    M: .word 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0,
     0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0,
       0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1
    T: .word 3, 0, 2, 0, 0, 0, 1, 0, 3
    # --- End no modificar labels---
    # de aca para abajo van sus variables en memoria
.text
    start:
        # do stuff
```

Se evaluará que se respete la convención de llamada y que se entregue un *output* correcto a partir de una serie de casos de prueba.

Tarea 3 - Revisión Pregunta 1

Ejercicio en clases

Desarrolle un programa en RISC-V que determine si un *input N* definido en el segmento `.data` es primo o no. El programa debe imprimir un 1 si lo es y un 0 en caso contrario.

Antes de terminar

¿Dudas?

¿Consultas?

¿Inquietudes?

¿Comentarios?

DCC

DEPARTAMENTO DE CIENCIA
DE LA COMPUTACIÓN

IIC2343

Arquitectura de Computadores

Clase 9 - Arquitectura RISC-V

Profesor: Germán Leandro Contreras Sagredo