Tarea 1: Arquitecturas RISC

1nd Christopher Rodríguez Cordero
Escuela de Ingeniería en Computadores
Instituto Tecnológico de Costa Rica
Cartago, Costa Rica
chrodriguez@estudiantec.cr

I. RISC-V RV32 (base RV32I): tipos de instrucciones y codificación

RISC-V es una ISA RISC abierta con formatos fijos de 32 bits en su base RV32I; *pues* esta base define seis formatos canónicos (R, I, S, B, U, J) que permiten cubrir operaciones aritmético–lógicas, memoria, saltos y sistema, *porque* la separación entre clases y formatos simplifica decodificación y verificación [1], [2]. En *cualuier caso*, la especificación mantiene compatibilidad hacia extensiones (M/A/F/D/C, etc.) sin alterar los formatos base, *pero* deja claro que la codificación de cada campo es estable y ratificada por RISC-V International [3]. *Entonces*, un compilador puede generar código portátil sobre cualquier implementación RV32I. *Es más*, la edición 20240411 consolida descripciones normativas y diagramas de bits útiles para verificación formal [1].

Tipos y ejemplos (RV32I)

- **R** (ALU registro–registro): `ADD`, `SUB`, `AND`, `OR`, `SLL`, `SLT`, etc.
- **I** (ALU inmediato, cargas, `JALR`, sistema): `ADDI`, `XORI`, `LW`, `CSRRS`, `ECALL`.
- **S** (almacenamientos): `SB`, `SH`, `SW`.
- **B** (ramas): `BEQ`, `BNE`, `BLT`, `BGEU`.
- **U** (inmediato alto): `LUI`, `AUIPC`.
- **J** (salto absoluto PC-relativo): `JAL`.

Resumen de codificación (31 .. 0)

Tabla I. RV32I: formatos y campos principales [1].

Formato	Disposición de bits
R	`funct7[31:25] | rs2[24:20] | rs1[19:15] | funct3[14:12] | rd[11:7] | opcode[6:0]`
I	`imm[31:20] | rs1 | funct3 | rd | opcode`
S	`imm[31:25] | rs2 | rs1 | funct3 | imm[11:7] | opcode`
B	`imm[31], imm[7], imm[30:25], rs2, rs1, funct3, imm[11:8], opcode`
U	`imm[31:12] | rd | opcode`
J	`imm[31], imm[19:12], imm[20], imm[30:21], rd, opcode`

Notas importantes: (i) todas las instrucciones base miden 32 bits; (ii) los inmediatos B/J están reordenados y son PC–relativos con `LSB=0`; (iii) `JALR` usa I y limpia `LSB` del destino; (iv) el conjunto comprimido C (opcional) introduce codificaciones de 16 bits [1].

II. ARM de 32 bits (ARMv7): tipos de instrucciones y codificación

ARMv7 define los perfiles A/R (aplicaciones y tiempo real) con conjunto ARM de 32 bits y Thumb-2 (16/32 bits); *pues* la ISA prioriza tanto rendimiento como densidad. Muchas instrucciones ARM llevan campo de condición `cond[31:28]`, *porque* la ejecución condicional amplia el control de flujo sin ramas adicionales; *en cualuier caso*, Thumb-2 reubica la condicionalidad (p. ej., `IT`) para preservar densidad [4], [5]. *Pero* ciertas operaciones (coprocesador/SIMD) dependen de implementación y perfil. *Entonces*, conviene distinguir entre codificación ARM y la de Thumb-2 al analizar binarios. *Es más*, el ARM ARM (DDI0406C) documenta exhaustivamente campos y modos [4], [6].

Tipos y ejemplos (ARM 32-bit)

- **Data processing**: `ADD`, `SUB`, `MOV`, `CMP`, con operandos registro o inmediato rotado; bit `S` puede actualizar flags.
- **Load/Store**: `LDR/STR` con base+desplazamiento, pre/post-indexado y *write-back*; variantes byte/half/signed.
- **Branch**: `B/BL/BLX` con desplazamiento PC–relativo con signo; `BL` enlaza a `LR`.
- **Multiply/MLA**, **LDM/STM** (transferencias múltiples), **MRS/MSR** (estado), y coprocesador/VFP/NEON (según perfil).

Esquema de codificación (ARM 32-bit)

En Thumb-2, instrucciones de 32 bits se forman por dos *halfwords*; varias clases replican funcionalidad de ARM con codificación distinta y reglas de alineación/ensamblado específicas [5].

Tabla II. ARMv7 (ARM 32-bit): patrones típicos [4].

Clase	Idea de campos (31 .. 0)
Data proc.	`cond | opcode | S | Rn | Rd | operand2 (inm rotado o Rm+shift)`
Load/Store	`cond | 01 | I | P | U | B | W | L | Rn | Rd | addr_mode`
Branch	`cond | 101 | L | imm24 (sign-extend, «2)`
Mult/MLA	`cond | 000000 | A | S | Rd | Rn | Rs | 1001 | Rm`

III. Programa en ensamblado

III-A. RISC-V

En la siguiente figura 1 vemos el código de ensamblador para RISC-V.

Figura 1. Codigo inicial

```
    .data
cadena: .ascii "cadena"

    .text
    .globl _start

_start:
    la t0, cadena       # t0 = dirección base de la cadena
    la t1, cadena       # t1 = dirección para encontrar final

# Longitud de la cadena
len_loop:
    lb t2, 0(t1)        # cargar byte
    beqz t2, len_found  # si es 0 -> fin
    addi t1, t1, 1
    j len_loop

len_found:
    addi t1, t1, -1     # t1 apunta al último carácter válido

# Invertir
reverse_loop:
    blt t1, t0, done    # si t1 < t0, fin

    lb t2, 0(t0)        # cargar byte de inicio
    lb t3, 0(t1)        # cargar byte de fin
    sb t3, 0(t0)        # escribir fin en inicio
    sb t2, 0(t1)        # escribir inicio en fin

    addi t0, t0, 1      # mover inicio adelante
    addi t1, t1, -1     # mover fin atrás
    j reverse_loop

done:
    # Loop infinito para detener ejecución
    j done
```

En la siguiente figura 2 se ve la cadena antes de ser invertida.

Figura 2. Cadena Original

```
00000000  04000293  04000313  00030383  00038663    ···· ···· ···· c···
00000010  00130313  ff5ff06f  fff30313  02534063    ···· o·_· ···· c@S·
00000020  00028383  00030e03  01c28023  00730023    ···· ···· #·s·
00000030  00128293  fff30313  fe5ff06f  0000006f    ···· ···· o·_· o···
00000040  65646163  0000616e  aaaaaaaa  aaaaaaaa    cade na·· ···· ····
00000050  aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    ···· ···· ···· ····
```

Y en la figura 3 se ve la cadena invertida.

Figura 3. Cadena Invertida

```
00000000  04000293  04000313  00030383  00038663    ···· ···· ···· c···
00000010  00130313  ff5ff06f  fff30313  02534063    ···· o·_· ···· c@S·
00000020  00028383  00030e03  01c28023  00730023    ···· ···· #··· #·s·
00000030  00128293  fff30313  fe5ff06f  0000006f    ···· ···· o·_· o···
00000040  64656e61  00006361  aaaaaaaa  aaaaaaaa    aned ac·· ···· ····
```

IV. Programa en ensamblado

IV-A. ARMv4

En la siguiente figura 14 vemos el código de ensamblador para ARMv4.

Figura 4. Codigo inicial

```
    .data
cadena: .asciz "cadena"

    .text
    .global _start

_start:
    LDR r0, =cadena     @ r0 = inicio
    LDR r1, =cadena     @ r1 = para calcular longitud

@ strlen
len_loop:
    LDRB r2, [r1]
    CMP r2, #0
    BEQ len_found
    ADD r1, r1, #1
    B len_loop

len_found:
    SUB r1, r1, #1      @ r1 apunta al último carácter

@ Invertir
reverse_loop:
    CMP r1, r0
    BLT done

    LDRB r2, [r0]       @ byte inicio
    LDRB r3, [r1]       @ byte fin
    STRB r3, [r0]       @ escribir fin en inicio
    STRB r2, [r1]       @ escribir inicio en fin

    ADD r0, r0, #1      @ mover inicio
    SUB r1, r1, #1      @ mover fin
    B reverse_loop
done:
    B done
```

En la siguiente figura 5 se ve la cadena antes de ser invertida.

Y en la figura 6 se ve la cadena invertida.

Figura 5. Cadena Original

```
00000000  e59f0040  e59f103c  e5d12000  e3520000    @···  <···  ·  ··  ··R·
00000010  0a000001  e2811001  eafffffa  e2411001    ····  ····  ····  ··A·
00000020  e1510000  ba000006  e5d02000  e5d13000    ··Q·  ····  ·  ··  ·0··
00000030  e5c03000  e5c12000  e2800001  e2411001    ·0··  ·  ··  ····  ··A·
00000040  eafffff6  eafffffe  00000050  00000000    ····  ····  P···  ····
00000050  65646163  0000616e  aaaaaaaa  aaaaaaaa    cade na··  ····  ····
```

Figura 6. Cadena Invertida

```
00000000  e59f0040  e59f103c  e5d12000  e3520000    @···  <···  ·  ··  ··R·
00000010  0a000001  e2811001  eafffffa  e2411001    ····  ····  ····  ··A·
00000020  e1510000  ba000006  e5d02000  e5d13000    ··Q·  ····  ·  ··  ·0··
00000030  e5c03000  e5c12000  e2800001  e2411001    ·0··  ·  ··  ····  ··A·
00000040  eafffff6  eafffffe  00000050  00000000    ····  ····  P···  ····
00000050  64656e61  00006361  aaaaaaaa  aaaaaaaa    aned ac··  ····  ····
```

V. Comparativa de características: RISC-V (RV32) vs. ARMv4

Endianness

RISC-V define little-endian por defecto a nivel de ISA; *pues* la base especifica el orden de bytes para accesos de 8/16/32 bits, *porque* uniformiza el entorno de ejecución y ABI. *En cualuier caso*, existen variantes big/bi-endian dependientes de plataforma pero no son el modo dominante [1]. ARMv4, por su parte, es bi-endian: soporta big y little con reglas de mapeo y alineación históricas; *pero* detalles como el "word-invariant big-endian" y controles de configuración evolucionaron hasta ARMv7 [6], [7]. *Entonces*, en software embebido clásico con ARM7TDMI se encuentran ambos esquemas según SoC/firmware. *Es más*, la documentación moderna de Arm mantiene la consistencia de little-endian como modo por defecto en muchas toolchains [6].

Direccionamiento (memoria y saltos)

En RV32, los accesos usan base+inmediato de 12 bits (I/S), memoria direccionada por byte y alineación recomendada; las ramas/saltos son PC–relativos con inmediatos reordenados (B/J), y `AUIPC+JALR` permiten saltos largos/indirectos [1]. *Pues* esta simplicidad favorece pipelines limpios, *porque* el hardware de cálculo de direcciones es uniforme. *En cualuier caso*, las extensiones no cambian la semántica base. En ARMv4,

hay múltiples modos de direccionamiento: desplazamientos por registro con *shift*, pre/post-indexado, *write-back*, transferencias múltiples (`LDM/STM`) y ramas PC–relativas con enlace a `LR`; *pero* la mayor flexibilidad implica más lógica de decodificación [4], [8]. *Entonces*, optimizaciones clásicas en ARMv4 explotan modos complejos para reducir instrucciones de *load/store*. *Es más*, esta filosofía contrasta con la minimalista de RV32.

V-A. Anexos

En el siguiente link: https://github.com/ChrisRC7/Tarea-1.-Arquitecturas-RISC encontrará el repositorio donde encontrará este mismo pdf y ambos archivos con el código de ensamblador.

Referencias

[1] *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA (Version 20240411)*, Accessed 2025-08-15, RISC-V International, 2024. dirección: https://courses.grainger.illinois.edu/ece391/su2025/docs/unpriv-isa-20240411.pdf.

[2] A. Waterman, Y. Lee, D. A. Patterson y K. Asanović, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 2.1," EECS Department, University of California, Berkeley, inf. téc. UCB/EECS-2016-118, 2016, Accessed 2025-08-15. dirección: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.pdf.

[3] "RISC-V Ratified Specifications." Accessed 2025-08-15, RISC-V International. (2025), dirección: https://riscv.org/specifications/ratified/.

[4] *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition (DDI0406C)*, Accessed 2025-08-15, Arm Limited, 2012. dirección: https://developer.arm.com/documentation/ddi0406/latest/.

[5] *Armv7-M Architecture Reference Manual (DDI0403)*, Accessed 2025-08-15, Arm Limited, 2025. dirección: https://developer.arm.com/documentation/ddi0403/latest/.

[6] "ARM Architecture Reference Manual ARMv7-A and ARMv7-R." Accessed 2025-08-15, Arm Limited. (2025), dirección: https://developer.arm.com/documentation/ddi0406/cb.

[7] "Endian support in ARMv4/ARMv5 and ARMv7." Accessed 2025-08-15, Arm Developer. (2025), dirección: https://developer.arm.com/documentation/ddi0406/cb/Appendixes/ARMv4-and-ARMv5-Differences/Application-level-memory-support/Endian-support.

[8] *ARM Architecture Reference Manual (legacy) – Addressing Modes and Instruction Set Overview*, Accessed 2025-08-15, Arm Limited, 2007. dirección: https://www.macs.hw.ac.uk/~hwloidl/Courses/F28HS/Docu/DDI0406C_C_arm_architecture_reference_manual.pdf.