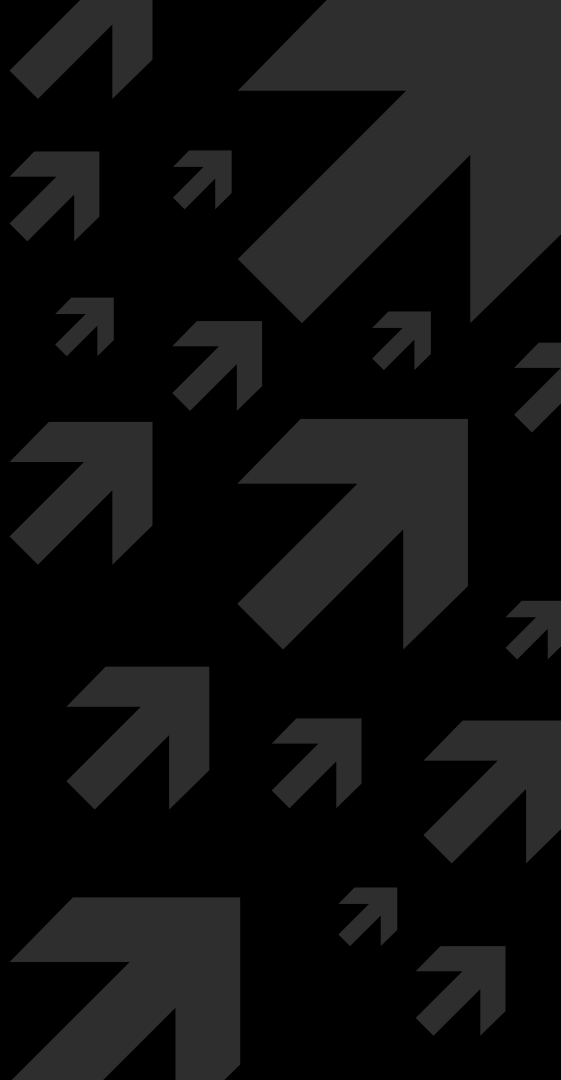
RISC-V

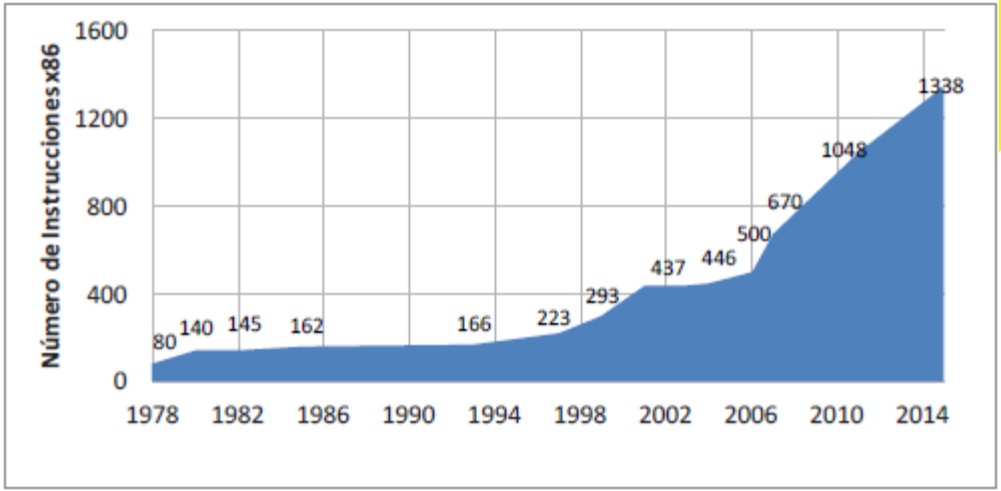
1. RISC-V

Historia



Problemática...

- Las ISA existentes eran complejas y con derechos de propiedad.
- Partiendo del procesador Intel 8086, el set x86, parte con 80 instrucciones en 1978.
- Para el 2014, alcanzó las 1338 instrucciones en su versión x86 32 bits.

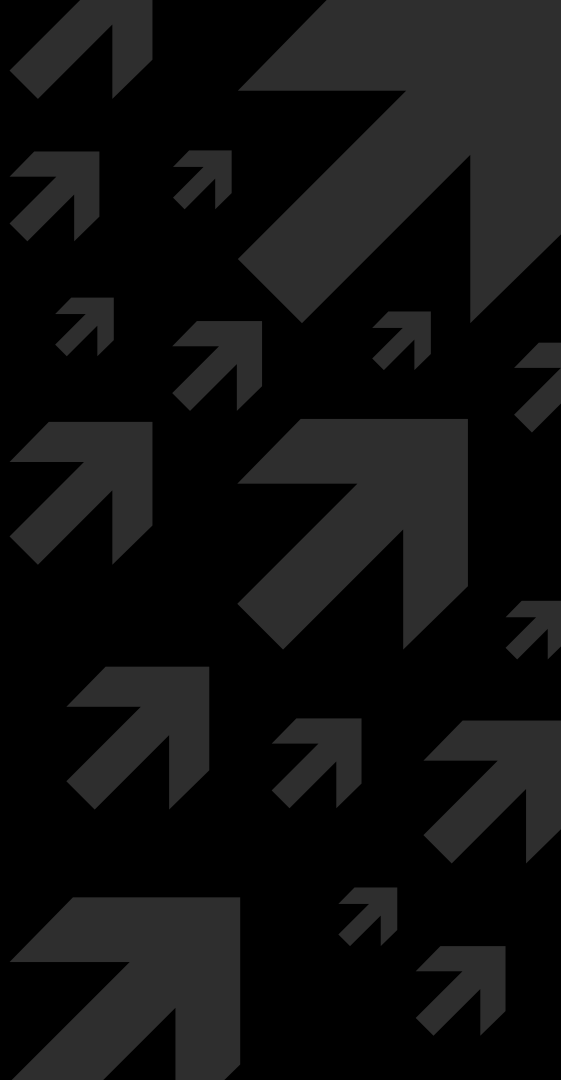


Origen

- RISC (Reduced Instruction Set Computer) inició como proyecto temporal en UC Berkeley.
- Surge de la necesidad de una ISA libre y cambiando la tendencia incremental de las ISAs existentes a una modular, con un núcleo fundamental y adaptándolo a las necesidades específicas.
- El núcleo fundamental del ISA de RISC-V es llamado **RV32I.**
- Las extensiones se indican mediante banderas representadas mediante concatenación al núcleo fundamental
  - Ejemplo: RV32IMF, agrega multiplicación RV32M y punto flotante precisión simple RV32F, a las instrucciones base obligatorias RV32I).

2. RISC-V

Compilación



Ensamblador

- Recibe como entrada un archivo con extensión “.s” en UNIX y un archivo con extensión “.asm” para MS-DOS .
- El ensamblador no solo produce código objeto, si no que extiende las instrucciones para incluir operaciones útiles, por lo que se les llama “pseudoinstrucciones”.
- Puede incluir instrucciones de alto nivel, como llamadas a librerías incluidas particulares.

Enlazador

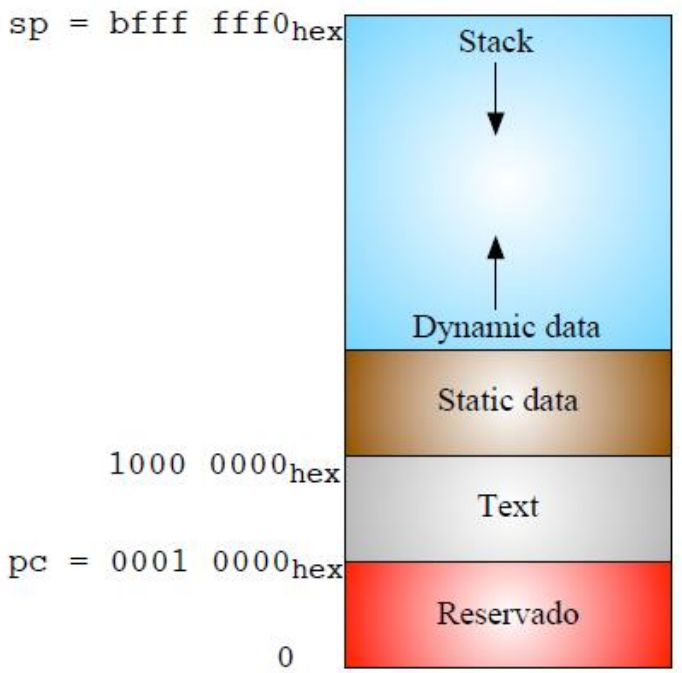
- Recibe como entrada un archivo con extensión “.o” en UNIX, generando un archivo de salida “.out”; para MS-DOS, recibe un archivo con extensión “.obj” o “.lib” y genera un archivo de salida “.exe”.

- El enlazador permite que, en vez de compilar todo el código fuente cada vez que se cambia un archivo, se pueda ensamblar archivos individuales por separado.

- Puede incluir instrucciones de alto nivel, como llamadas a librerías incluidas particulares.

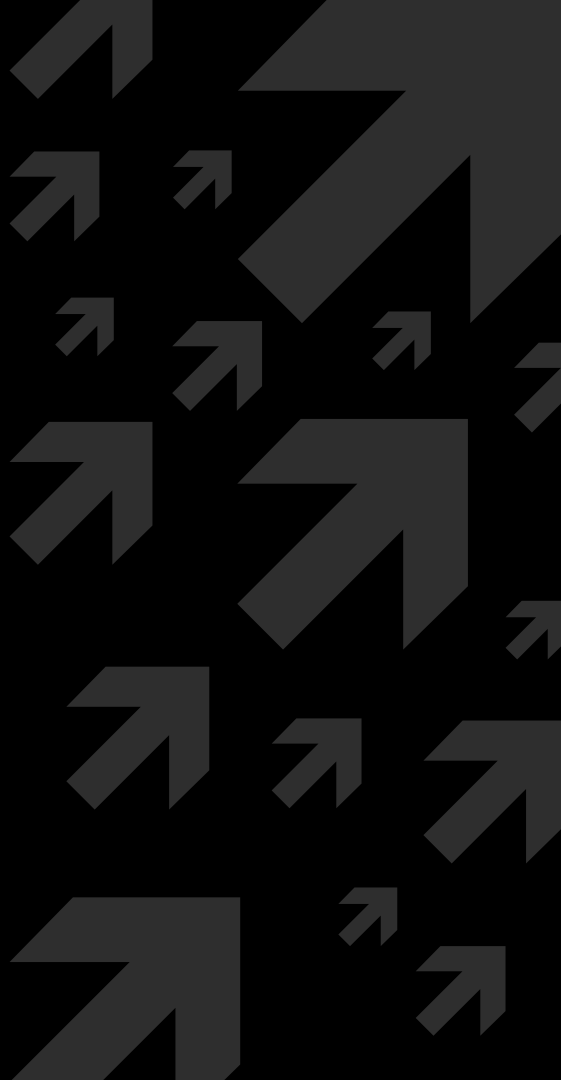
Memoria

- El **stack pointer** (sp) comienza en bfff fff0 y crece hacia el área de Static data.
- El área de **text** (código del programa) comienza en 0001 0000 e incluye librerías enlazadas en forma estática.
- El área de **Static data** comienza inmediatamente después del área de text (ejemplo: dirección 1000 0000).
- Los datos dinámicos están después del Static data, llamado Heap, crece hacia el área de stack e incluye librerías enlazadas en forma dinámica.



3. RISC-V

Directivas



Directivas en RV32I

- Inician por “.” y luego el identificador.
- Instrucciones predefinidas para la habilitación de elementos específicos del programa.
- No solo se reservan para la creación de segmentos si no permiten la definición de almacenamientos en memoria.

Directiva	Descripción
`.text`	Ítems subsiguientes son almacenados en la sección text (código de máquina).
`.data`	Ítems subsiguientes son almacenados en la sección data (variables globales).
`.bss`	Ítems subsiguientes son almacenados en la sección bss (variables globales inicializadas a 0).
`.section .foo`	Ítems subsiguientes son almacenados en la sección llamada `.foo`.
`.align n`	Alinear el siguiente dato en un límite de 2^n bytes. Por ejemplo, `.align 2` alinea el próximo valor en un límite de *word*.
`.balign n`	Alinear el siguiente dato en un límite de n bytes. Por ejemplo, `.balign 4` alinea el próximo valor en un límite de *word*.
`.globl sym`	Declara que la etiqueta sym es global y se le puede hacer referencia desde otros archivos.
`.string "str"`	Almacenar el string str en memoria y terminarlo en null.
`.byte b1,..., bn`	Almacenar las n cantidades de 8 bits en bytes sucesivos de memoria.
`.half w1,...,wn`	Almacenar las n cantidades de 16 bits en halfwords sucesivos de memoria.
`.word w1,...,wn`	Almacenar las n cantidades de 32 bits en words sucesivos de memoria.
`.dword w1,...,wn`	Almacenar las n cantidades de 64 bits en doublewords sucesivos de memoria.
`.float f1,..., fn`	Almacenar los n números de punto flotante de precisión simple en words sucesivos de memoria.
`.double d1,..., dn`	Almacenar los n números de punto flotante de precisión doble en double-words sucesivos de memoria.
`.option rvc`	Comprimir las instrucciones subsiguientes (ver Capítulo 7).
`.option norvc`	No comprimir las instrucciones subsiguientes.
`.option relax`	Permitir relajación del linker para las instrucciones subsiguientes.
`.option norelax`	No permitir relajación del linker para las instrucciones subsiguientes.
`.option pic`	Las instrucciones subsiguientes son *position-independent code*.
`.option nopic`	Las instrucciones subsiguientes son *position-dependent code*.
`.option push`	Hacer Push del estado de todos los `.options` a un stack, para que un posterior `.option pop` restaure sus valores.
`.option pop`	Hacer Pop del stack de opciones, restaurando todos los `.options` a su estado en el momento del último `.option push`.

Directivas comunes de RISC-V

4. RISC-V

Registros

Registros de RV32I

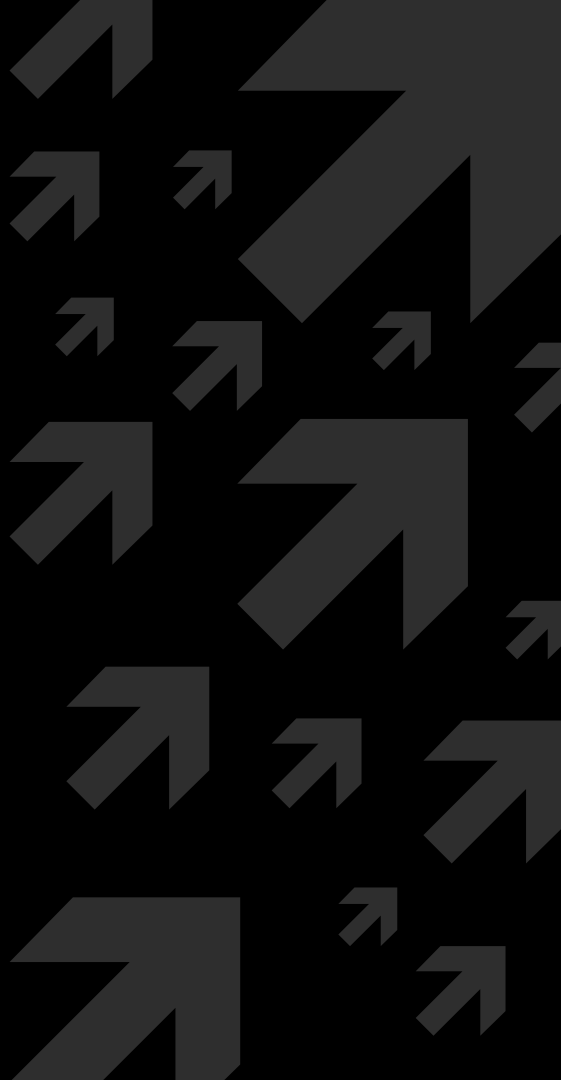
- Sus nombres están determinados por el **ABI** (Application Binary Interface).
- En RV32I existen **31 registros** utilizables, más el **x0** que siempre tiene el valor 0.
- Todos los registros y tamaños de palabras son de **32 bits**.
- Todas las operaciones son entre registros (ninguna de registro a memoria).

00000	x0	**zero**	Hard-wired zero	
00001	x1	**ra**	Return address Caller	May be changed by sub Used for function return with jalr x0, x1, 0
00010	x2	**sp**	Stack pointer Callee	if changed by Sub must be backed up
00011	x3	**gp**	Global pointer	
00100	x4	**tp**	Thread pointer	
00101	x5	**t0**	Temporary/alternate link register Caller	May be changed by sub
00110	x6	**t1**	Temporaries Caller	May be changed by sub
00111	x7	**t2**	Temporaries Caller	May be changed by sub
01000	x8	**s0 / fp**	Saved register/frame pointer Callee	if changed by Sub must be backed up
01001	x9	**s1**	Saved register Callee	if changed by Sub must be backed up
01010	x10	**a0**	Function arguments/return values Caller	Use to pass to functions - May be changed by sub
01011	x11	**a1**	Function arguments/return values Caller	Use to pass to functions - May be changed by sub
01100	x12	**a2**	Function arguments Caller	Use to pass to functions - May be changed by sub
01101	x13	**a3**	Function arguments Caller	Use to pass to functions - May be changed by sub
01110	x14	**a4**	Function arguments Caller	Use to pass to functions - May be changed by sub
01111	x15	**a5**	Function arguments Caller	Use to pass to functions - May be changed by sub
10000	x16	**a6**	Function arguments Caller	Use to pass to functions - May be changed by sub
10001	x17	**a7**	Function arguments Caller	Use to pass to functions - May be changed by sub
10010	x18	**s2**	Saved registers Callee	if changed by Sub must be backed up
10011	x19	**s3**	Saved registers Callee	if changed by Sub must be backed up
10100	x20	**s4**	Saved registers Callee	if changed by Sub must be backed up
10101	x21	**s5**	Saved registers Callee	if changed by Sub must be backed up
10110	x22	**s6**	Saved registers Callee	if changed by Sub must be backed up
10111	x23	**s7**	Saved registers Callee	if changed by Sub must be backed up
11000	x24	**s8**	Saved registers Callee	if changed by Sub must be backed up
11001	x25	**s9**	Saved registers Callee	if changed by Sub must be backed up
11010	x26	**s10**	Saved registers Callee	if changed by Sub must be backed up
11011	x27	**s11**	Saved registers Callee	if changed by Sub must be backed up
11100	x28	**t3**	Temporaries Caller	May be changed by sub
11101	x29	**t4**	Temporaries Caller	May be changed by sub
11110	x30	**t5**	Temporaries Caller	May be changed by sub
11111	x31	**t6**	Temporaries Caller	May be changed by sub

Registros de RV32I

5. RISC-V

Instrucciones



Instrucciones de Movimiento

li rd, immediate
Load Immediate. Pseudoinstrucción, RV32I y RV64I.

```
x[rd] = immediate
```

la rd, symbol
Load Address. Pseudoinstrucción, RV32I y RV64I.

```
x[rd] = &symbol
```

mv rd, rs1
Move. Pseudoinstrucción, RV32I y RV64I.

```
x[rd] = x[rs1]
```

Instrucciones de Salto

j offset

Jump. Pseudoinstrucción, RV32I y RV64I.

Descripción	RV32I	ARM-32	x86-32
Branch si $=$	`beq t0, t1, foo`	`cmp r0, r1` `beq foo`	`cmp eax, esi` `je foo`
Branch si $\neq$	`bne t0, t1, foo`	`cmp r0, r1` `bne foo`	`cmp eax, esi` `jne foo`
Branch si $<$	`blt t0, t1, foo`	`cmp r0, r1` `blt foo`	`cmp eax, esi` `jl foo`
Branch si $\geq_s$	`bge t0, t1, foo`	`cmp r0, r1` `bge foo`	`cmp eax, esi` `jge foo`
Branch si $<_u$	`bltu t0, t1, foo`	`cmp r0, r1` `bcc foo`	`cmp eax, esi` `jb foo`
Branch si $\geq_u$	`bgeu t0, t1, foo`	`cmp r0, r1` `bcs foo`	`cmp eax, esi` `jnb foo`
Branch si $=0$	`beqz t0, foo`	`cmp r0, #0` `beq foo`	`test eax, eax` `je foo`
Branch si $\neq 0$	`bnez t0, foo`	`cmp r0, #0` `bne foo`	`test eax, eax` `jne foo`

Instrucciones de Operación

add rd, rs1, rs2 — `x[rd] = x[rs1] + x[rs2]`

Add. Tipo R, RV32I y RV64I.
Suma el registro x[*rs2*] al registro x[*rs1*] y escribe el resultado en x[*rd*]. Overflow aritmético ignorado.

addi rd, rs1, immediate — `x[rd] = x[rs1] + sext(immediate)`

Add Immediate. Tipo I, RV32I y RV64I.
Suma el *inmediato* sign-extended al registro x[*rs1*] y escribe el resultado en x[*rd*]. Overflow aritmético ignorado.

sub rd, rs1, rs2 — `x[rd] = x[rs1] - x[rs2]`

Subtract. Tipo R, RV32I y RV64I.
Subtracts register x[*rs2*] from register x[*rs1*] y escribe el resultado en x[*rd*]. Overflow aritmético ignorado.

Instrucciones de Operación

and rd, rs1, rs2 — `x[rd] = x[rs1] & x[rs2]`

AND. Tipo R, RV32I y RV64I.
Calcula el AND a nivel de bits de los registros x[*rs1*] y x[*rs2*] y escribe el resultado en x[*rd*].

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*].

or rd, rs1, rs2 — `x[rd] = x[rs1] | x[rs2]`

OR. Tipo R, RV32I y RV64I.
Calcula el OR inclusivo a nivel de bits de los registros x[*rs1*] y x[*rs2*] y escribe el resultado en x[*rd*].

ori rd, rs1, immediate — `x[rd] = x[rs1] | sext(immediate)`

OR Immediate. Tipo I, RV32I y RV64I.
Calcula el OR inclusivo a nivel de bits del *inmediato* sign-extended y el registro x[*rs1*] y escribe el resultado en x[*rd*].

not rd, rs1 — `x[rd] = ~x[rs1]`

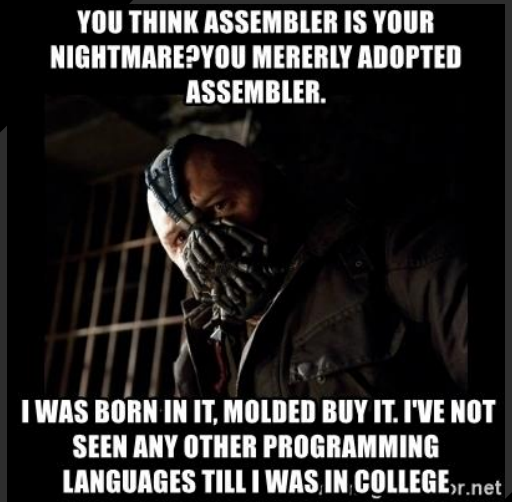
NOT. Pseudoinstrucción, RV32I y RV64I.

Instrucciones de Interrupción

ecall

Environment Call. Tipo I, RV32I y RV64I.

¿Dudas?


YOU THINK ASSEMBLER IS YOUR NIGHTMARE?YOU MERERLY ADOPTED ASSEMBLER.
I WAS BORN IN IT, MOLDED BUY IT. I'VE NOT SEEN ANY OTHER PROGRAMMING LANGUAGES TILL I WAS IN COLLEGE