

Lenguajes de programación 62-2025-25
 Tarea 09 - Diseño de un computador propio inspirado en el
 computador del Profesor Peña

Integrantes

- 1 Camilo Andrés Medina Seánchez
- 2 María Camila Castro Pernas
- 3 Iván David Boitrago Salazar
- 4 Daniela Adriana Rueda Hernández
- 5 Christian Steven Matta Ojeda
- 6 Luis David Garzón Morales.

Enunciado:

1. Inspirarse en la sección 2.5 cómo funciona un computador del libro de Euclides a Java.
2. Realice los diseños completos de un computador propuesto bajo la arquitectura John von Neumann

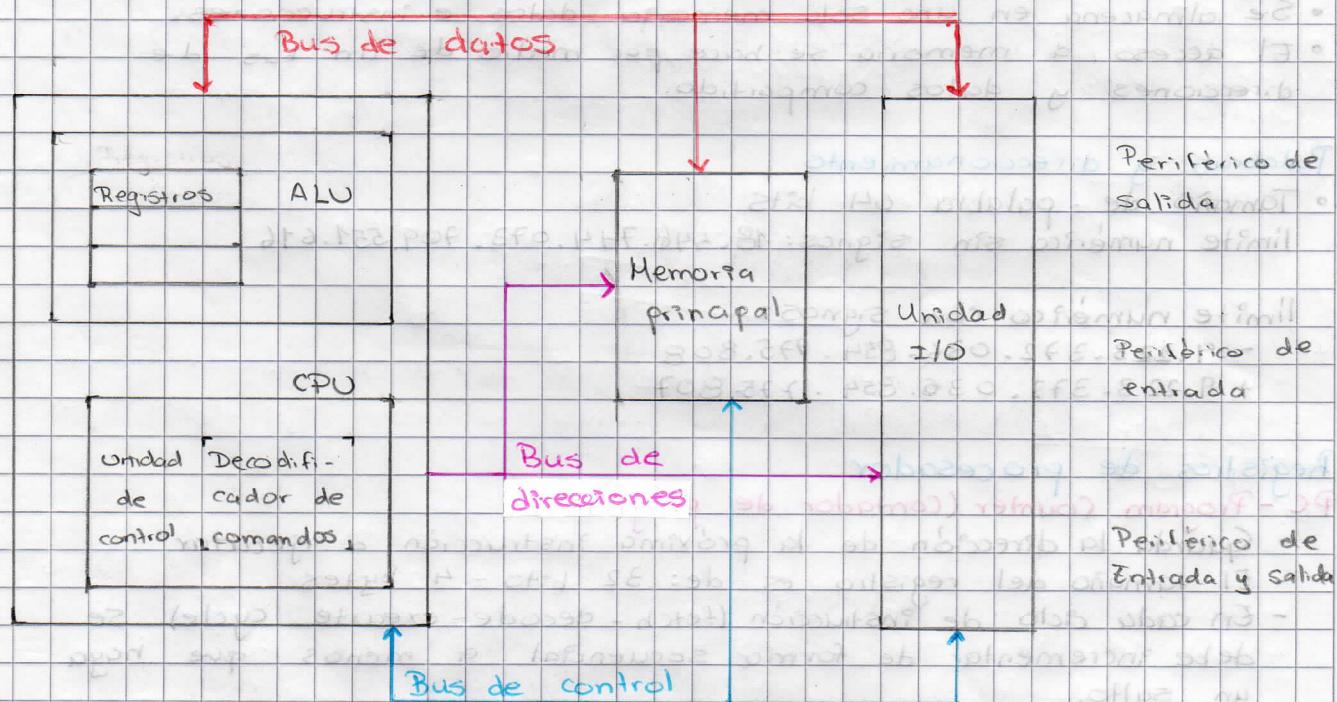


Diagrama de arquitectura John von Neumann

3. Todos los módulos de la figura deben ser funcionales.
 La palabra del computador debe ser de 8 bytes (64 bits).
 Se debe indicar los formatos de microinstrucciones con 40 de estas.
4. Diseñar el módulo Enlazador-Cargador.
5. Seleccione tres algoritmos clásicos, escribalos en su ensamblaje, haga manualmente su traducción al lenguaje máquina de su computador y verifique que la máquina está en la capacidad de

resolver problemas relativamente complejos.

6. Diseñe un pequeño sistema operativo (opcional).

7. Dele nombre a la empresa y al computador.

Nombres

Empresa: Peñatech Labs

Computador: Euclid - 64

Arquitectura base

Modelo de arquitectura: Von Neumann

Tamaño de palabra: 64 bits

Endiannes: Little - Endian

Diseño de arquitectura

- John Von Neumann:
- Se almacena en una sola memoria datos e instrucciones.
- El acceso a memoria se hace por medio de un bus de direcciones y datos compartido.

Palabras y direccionamiento

- Tamaño de palabra 64 bits

Límite numérico sin signos: 18.446.744.073.709.551.616

Límite numérico con signos:

- 9.223.372.036.854.775.808

+ 9.223.372.036.854.775.807

Registros de procesador

PC - Program Counter (Contador de programa).

- Guarda la dirección de la próxima instrucción a ejecutar
- El tamaño del registro es de: 32 bits - 4 bytes
- En cada ciclo de instrucción (fetch - decode - execute cycle) se debe incrementar de forma secuencial a menos que haya un salto.
- En cada ciclo de instrucción el contenido del contador de programa se envía al Memory Address Register - MAR. (Este procedimiento se explicará más a fondo cuando se explique el ciclo de ejecución fetch - decode - execute).

IR - Instruction Register (Registro de instrucción).

- Guarda la instrucción que se está ejecutando actualmente
- Sus bits son enviados a la unidad de control para decodificarlos y obtener opcode y operandos (opcional).
- Permite que el contador de programa avance mientras guarda la instrucción actual.
- El tamaño del registro es de: 64 bits - 8 bytes

HAR - Memory Address Register (Registro de dirección de memoria)

- Contiene la dirección de memoria a la cual se desea acceder
- El acceso puede ser de tipo lectura o escritura.
- El HAR tiene una interacción directa con el bus de direcciones
- El tamaño del registro es de: 32 bits - 4 bytes

MDR - Memory Data Register (Registro de datos de memoria)

- Almacena el dato que entra o sale de memoria
- Como la memoria de acceso aleatoria dinámica (d-ram) contiene tanto instrucciones (opcode) como operandos. Entonces, este registro tiene una interacción directa con el IR y los registros de propósito general
 - Operando transfiere a reg. de propósito general
 - Instrucción transfiere a instrucción register
- El tamaño del registro es de: 64 bits - 8 bytes

ACUM - Acumulator (Acumulado)

- Registro para uso de la unidad aritmético-lógica (ALU).
- Almacena resultado de operaciones
- El tamaño del registro es de: 64 bits - 8 bytes.

Flag Register / status register / condition code register

El flag register es un registro especial que indica el estado de la unidad aritmético-lógica (ALU).

Para el propósito de este modelo lógico académico el flag register será de ocho bits como se muestra a continuación

0	1	2	3	4	5	6	7
Z	C	N	P	O	I		

1. Zero (Z)

- Valor en 1 si el resultado de la última operación en la ALU es cero

2. Carry (C)

- Valor en 1 si en la última suma o resta ejecutada por la ALU se generó un carry.

3. Negative (N)

- Valor en 1 si el bit más significativo del resultado es 1.
- Número negativo en complemento a 2.

4. Positive (P)

- Valor en 1 si el resultado de la operación ejecutada por la ALU es positivo.
- Es un registro adicional puesto que se puede deducir a partir de $N=0$ y $Z=0$.

5. Overflow (O)

- Valor en 1 si ocurre un desbordamiento aritmético con signo (excede el rango máximo en 64 bits con signo).

6. Interrupt (I)

- Decidir si las señales externas están habilitadas
- Puede llegar a ser útil en procesos de I/O.

Registros de propósito general (GPRs)

A diferencia de los registros de procesador o los registros bandera (flag register), estos pueden tener un uso más

versatil en el proceso de ejecución.

Es común ver divisiones en donde se indica en que registros se debe salvar el valor de una función, el puntero de pila (stack pointer) o el puntero de base (base pointer). No obstante, para efectos prácticos y simplificar el diseño y planteamiento de las instrucciones que ejecuta la máquina se establece que se puede dar un uso a conveniencia de los 16 registros de propósito general (R0 - R15).

Mapa de memoria RAM / ROM / MMIO

A continuación se detalla un mapa que separa las direcciones posiblemente utilizables en términos físicos y las separaciones lógicas.

0x00000000	lógicas.
ROM	1.
268 mb	Firmware o bootloader que ejecuta rutinas principales al momento de encender la máquina. 0x00000000 - 0x0FFFFFFF
0x10000000	2.
RAM	Guardado de instrucciones a ejecutar, datos y operandos. 0x10000000 - 0x1FFFFFFF
268 mb - 4026 mb	3.
0x10000000	MMIO (Memory-Mapped I/O)
4026 mb - 4294 mb	Registros de dispositivos de entrada y salida. 0x1E000000 - 0xFFFFFFFF
0xFFFFFFF	

Espacio máximo direccionable en RAM

Tamaño bus de direcciones: 32 bits, 2^{32} bytes direccionables.

Tamaño de RAM asignada:

Rango: $0x10000000 \rightarrow 0x1FFFFFFF$

$$(0x1FFFFFFF - 0x10000000) + 1 = 3.758.096.384$$

$$\frac{3.758.096.384}{8} = 469\ 762\ 048 \text{ palabras de 64 bits.}$$

Mapa de interrupciones

Las interrupciones (interrupt request [IRQ]) son señales que pueden ser dadas desde hardware o software y que alteran el flujo de ejecución del procesador.

En términos generales, al recibir una solicitud de interrupción el procesador debe guardar su estado actual (PC, reg, flags) y saltar a una rutina especial ISR (interrupt service routine).

La ISR define la forma en que se deben generar las interrupciones.

Flujo que se debe seguir con una interrupción.

1. Ejecución normal de procesador.
2. Evento genera IRQ
3. La unidad de control verifica que las interrupciones estén habilitadas, Bit 5 del flag register [I].
4. CPU guarda su estado actual (Program counter, flags y registros de propósito general en uso activo).

5. El program counter (pc) se carga con la primera instrucción de la ISR
6. Ejecución de la ISR
7. Retorno a flujo de ejecución normal - **IRET**
8. Continua el flujo de ejecución.

El esquema de interrupciones planteado permite atender eventos como lo son:

- Externos: Eventos de entrada y salida.
- Internos: Excepciones
- Software: Establecidos en lenguaje de programación.

Definición de las IRQ a nivel Hardware.

IRQ 00	: Reset / inicio
IRQ 01	: Timer
IRQ 02	: Teclado
IRQ 03	: División por cero
IRQ 04	: Señal de periférico genérica
IRQ 05	: Overflow aritmético.

Ciclo de instrucción (fetch - decode - execute).

1. Fetch (Busqueda de la instrucción).

- 1.1. El program counter contiene la dirección de la próxima instrucción
- 1.2. Se accede desde el bus de direcciones a la memoria RAM
- 1.3. Por el bus de datos se regresan los 64 bits obtenidos del almacenamiento volátil.
- 1.4. La instrucción se almacena en el instruction register
- 1.5. Incremento secuencial del program counter para apuntar a la próxima dirección.

2. Decode (Decodificación de la instrucción).

- 2.1. El decodificador de instrucciones de la unidad de control toma el contenido del instruction register.

2.2. Separación de los campos de la instrucción

- opcode
- operandos
- valores
- registros

2.3. Se prepara para la ejecución de la instrucción

- Aritmética o lógica: Activación de ALU.
- Memoria: Prepara acceso de lectura/escritura en RAM o en MMIO.
- Salto: Ajustes del program counter.

3. Execute (Ejecución).

3.1. Operación aritmética / lógica

- Los operandos se cargan en ALU.
- El resultado se guarda en el ACUM.
- Actualización de flag register.

3.2. Acceso a memoria (LOAD / STORE)

- Load: La dirección se pone en MAR y el dato obtenido en el MDR, de ser necesario el valor pasa a un registro de propósito general, acumulador o si es instrucción pasa al instruction register (IR).
- Store: Del acumulador se pasa el valor al memory data register (MDR) y se guarda en la dirección contenida en MAR.

3.3. Instrucciones de salto, branch y condicionales

- El program counter se actualiza con la dirección de destino.

3.4. Interrupciones (IRQ)

- Se guarda la información necesaria del momento actual
- El PC se carga según el vector de interrupción en ROM con la dirección para ejecutar la ISR y manejar la IRQ

Señales de control

Las señales de control son líneas que permiten coordinar el flujo de ejecución en el computador. Estas son gestionadas por la unidad de control; Las señales de control no deben ser confundidas con las instrucciones en lenguaje ensamblador, teniendo en cuenta que estas están a un nivel más bajo y son las que permiten la ejecución de código ensamblador.

Por ejemplo, una operación de cargado a un registro

LOAD R1, [0x000F]

Se busca cargar en el registro 1 el valor de la posición de memoria **0x000F**

El proceso que se debe llevar a cabo desde las señales de control es como prosigue:

- Colocar el contenido del contador de programa (PC) en el registro de instrucciones (ir).
- Acceder al dato en RAM ubicado en **0x000F** (usando MAR)
- Este dato se debe guardar en el MDR
- Incrementar de forma secuencial el program counter (PC)
- El dato que está en el Memory Data Register se debe guardar en el RM.

Como se puede ver las señales de control son de crucial importancia al plantear el funcionamiento de las instrucciones según el opcode y definir el protocolo de ejecución de estos.

Descripción de las señales de control necesarias.

- CLK - Reloj de sistema (sincronización de estados).
- RESET - Reinicio de la unidad central de procesamiento
- IRQ[#] - Interruption request
- IRQACK - Reconocimiento de la irq.
- WAIT - Ciclos de espera
- MEMRD / MEMWR - Operación de lectura o escritura MMIO o RAM

- **HARLOAD/HAROUT**: Cargar el memory address register o pasar su información al bus de direcciones.
- **MDRLOAD/MDROUT**: Cargar el memory data register o pasar su información al bus de datos.
- **IRLOAD**: Cargar el ir con el contenido del memory data register.
- **PCINC**: Incremento en uno del program counter.
- **PCLOAD**: Cuando se necesita desarrollar una instrucción de salto, el program counter no se incrementa de forma secuencial. En cambio, se necesita cargar una nueva dirección.
- **REGRD[R#]**: Lectura del contenido de un registro de propósito general.
- **REGWR[R#]**: Cargar información en un registro de propósito general.
- **ACULOAD**: Cargar información en el registro Acumulador.
- **ALUOPERATION [##]**: Indicar la operación que debe realizar la ALU.
- **FLAGSLOAD**: Cargar el estado de la ALU en el registro de estado.
- **INTENABLE**: Habilitar o deshabilitar los IRQ.

Señal de control **ALUOPERATION [##]**:

000	: Adición
001	: Substracción
010	: Incremento
011	: Decremento
100	: AND
101	: OR
110	: XOR
111	: NOT

La siguiente tabla muestra los códigos disponibles para las operaciones aritméticas y lógicas de la ALU.

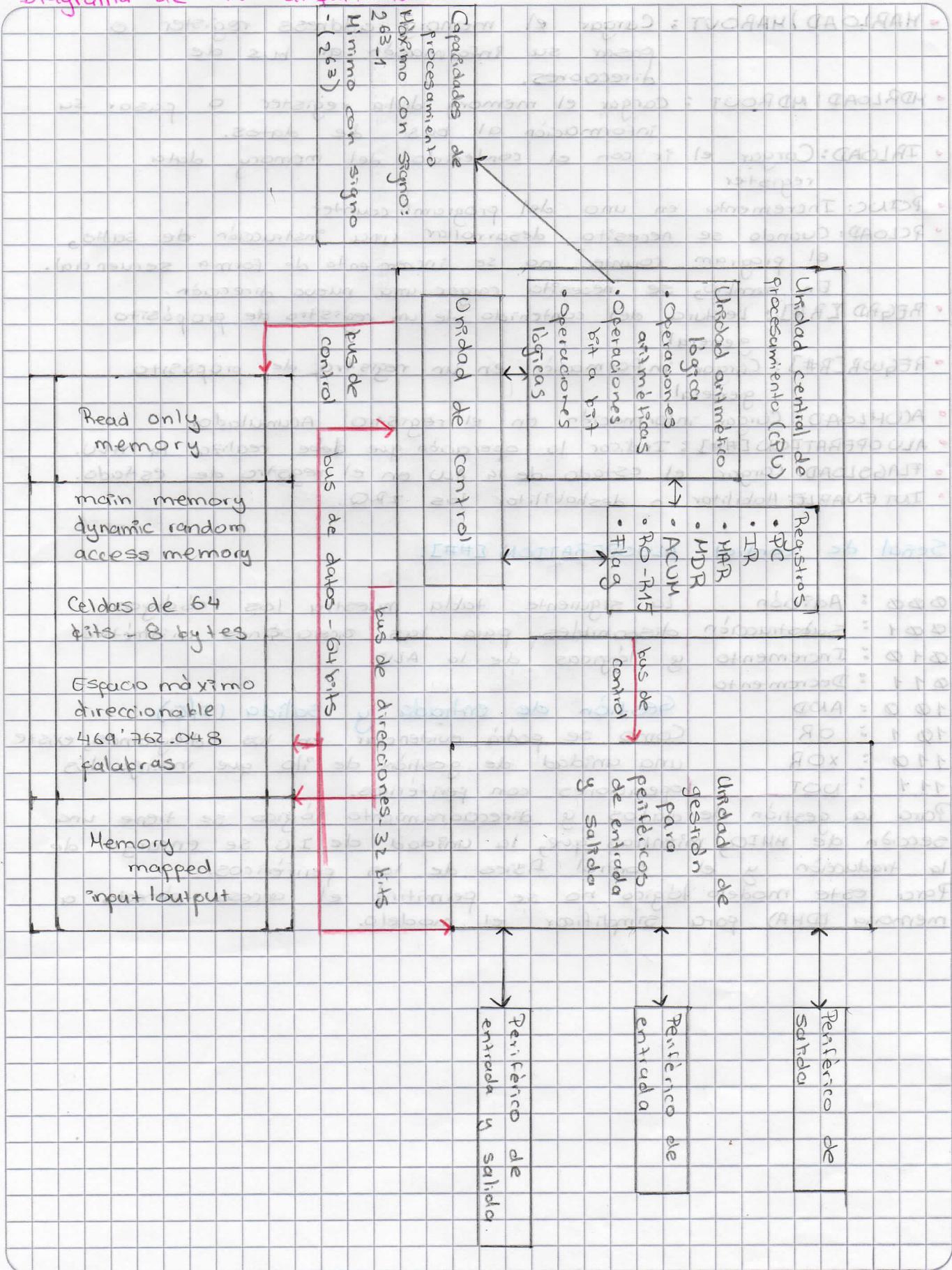
Gestión de entrada y salida (I/O).

Como se podrá evidenciar en los diagramas, existe una unidad de gestión de I/O que maneja las operaciones con periféricos.

Para la gestión de datos y direccionamiento lógico se tiene una sección de MMIO, mientras que, la unidad de I/O se encarga de la traducción y el control físico de los periféricos.

Para este modelo lógico no se permitirá el acceso directo a memoria (DMA) para simplificar el modelo.

Diagrama de la arquitectura



• Diseño de memorias:

• RAM:

- Organización: palabra = 64 bits
- Tamaño de ejemplo: 1 GiB
- Registro MAR: 32 bits
- Registro MDR: 64 bits
- Acceso: Lectura/escritura de 64 bits en un ciclo de bus
- Política: little-endian

• ROM:

- USO: microcodigo, rutinas de arranque, tabla vectores de interrupción y firmware base.
- Tamaño de ejemplo: 64 KiB
- Latencia: 1-2 ciclos

• Mapas físicos: Como se mencionó anteriormente:

- > ROM: $\emptyset \times \emptyset 00000000 - \emptyset \times \emptyset FF FFFF$
- > RAM: $\emptyset \times 1 00000000 - \emptyset \times E FFFFFFF F$
- > MMIO: $\emptyset \times F 00000000 - \emptyset \times FFFFFFF F$

• Buses: anchos y señales

• Buses físicos:

- Bus de datos: 64 bits, bidireccional
- Bus de direcciones: 32 bits, unidireccional ($CPU \rightarrow memoria$)
- Bus de control: señales de control principales:
 - * RD → active reading
 - * WR → active writing
 - * CS → chip select
 - * ACK → acknowledge
 - * CYC → in course bus cycle
 - * MMIO → access to peripheral map (optional)
 - * REQ_MASTER[n] / GNT_MASTER[n] → para arbitraje (n masters)
 - * INT_REQ[n] → petition for peripheral interruption.

• Señales Handshaking: señales de control de bus que coordinan correctamente las transacciones de lectura/escritura entre CPU, memoria y periféricos.

- * REQ / GNT
- * CYC + RD / WR
- * WAIT (optional)

• Protocolo de acceso a memoria: ($BADDR[31:0]$, $BDATA[63:0]$)

- Lectura:

1. CPU coloca dirección en $BADDR$, activos $CYC=1$ y $RD=F$
2. Si MMIO el dispositivo decodifica y prepara

dato; si es RAM, el controlador de memoria responde.

3. Cuando el dato está listo, dispositivo/memoria coloca el dato en BDATA y pone ACK=1.

4. CPU captura BDATA en MDR, baja RD y CYC; dispositivo baja ACK.

- Escritura:

1. CPU coloca BADDR y dato en BDATA, activa CYC=1 y WR=1.

2. Dispositivo/memoria detecta CS, guarda el dato.

3. Cuando termina, pone ACK=1. CPU limpia señales y continúa.

- Operaciones MMIO: mismas pasos que en mem access, pero la dirección se decodifica como peripheral → operaciones definidas por offsets de registros.

• Arbitraje:

- **Modelo propuesto:**
 > **arbiter centralizado**: con política configurable: round-robin por defecto o prioridad fija si se necesita tiempo real.

> **señales**: REQ-Master desde cada master; GNT-MASTER devuelto por arbiter.

> **LOCK** para operaciones multi-ciclo que requieren exclusividad

- Secuencia de arbitraje

1. Master solicita

2. arbiter decide

3. Master realiza la transacción.

4. Master libera GNT y REQ.

• Unidad de E/S (MMIO) y control de interrupciones

- **Tabla de registros**: Base: UART-BASE = 0xF000-0000

> 0×00 DATA (RD/WR) - lectura = recibir bytes, escritura = transmitir bytes.

> 0×08 STATUS (RD) - bit 0 = TX-READY, bit 1 = RX-READY, bit 7 = OVERRUN.

> 0×10 CONTROL (RD/WR) - bits enable TX/RX, parity.

> 0×18 IRQ-ENABLE (RD,WR) - habilita interrupciones para RX/TX.

FECHA

DIA

MES

AÑO

- controlador de interrupciones (PIC)

> Flujo IRQ

1. Periférico genera INT-REQ(n)
2. PIC detecta, determina vector VEC-n y acciona linea de petición al CPU
3. CPU, si IN-ENABLE = 1, completa el microciclo actual, guarda estado y salta a ISR en dirección vector-table [VEC-n]
4. ISR retorna con instrucción IRET que restaura estado.

• Tiempos de acceso y Latencias.

- ciclo base del bus: 10 ns
- RON: 1-2 ciclos según implementación
- MMIO: 1-3 ciclos (depende del periférico)
- operaciones: latencia variable y handshake con WAIT/ACK.

Hay que recordar que estos valores son tentativos, pues los valores reales dependen de la implementación física.

Formato de las instrucciones

Se ha escogido un formato de instrucción de longitud fija de 64 bits (una palabra) porque este diseño se ajusta de forma natural al ancho de palabra y al bus de datos de la máquina, ofrece uniformidad en la decodificación y en el tratamiento del contador de programa, y permite incluir en una sola palabra immediatos y direcciones de 32 bits sin fragmentación. La longitud fija favorece la simplicidad del ensamblador y del decodificador, reduce la complejidad de alineamiento y facilita la definición de convenciones claras para llamadas, pila y accesos a memoria; además asegura un equilibrio entre simplicidad pedagógica y expresividad (capacidad para codificar operaciones complejas y direcciones completas).

Diagrama de campos (bits 63..0) y fórmula de ensamblado:

OPCODE	RD	RS1	RS2	FUNC12	IMM32	0 bit
63	56	52	48	44	32	

$$f: \text{Word64} = (\text{OPCODE} \ll 56) | (\text{RD} \ll 52) | (\text{RS1} \ll 48) | (\text{RS2} \ll 44) | (\text{FUNC12} \ll 32) \\ | (\text{IMM32} \& 0xFFFFFFFF)$$

• **OPCODE** (Código de operación - 8 bits): Campo que identifica la instrucción completa; el decodificador toma OPCODE como primer criterio para seleccionar la semántica y la pauta de lectura de los restantes campos (familia funcional: ALU, memoria, control, E/S, sistema). Un espacio de 8 bits permite asignar un opcode distinto a la mayoría de las instrucciones y reservar rangos para futuras expansiones o grupos funcionales.

• **RD** (Registro destino - 4 bits): Codifica el registro (R0...R15) donde se deposita el resultado de la instrucción. En instrucciones sin destino efectivo (por ejemplo CMP) o cuando no se usa RD, por convención se escribe 0; este campo determina el operando de salida en las operaciones de registro y en muchas instrucciones inmediatas.

• **RS1** (Registro fuente 1 - 4 bits): Primer operando fuente codificado en 4 bits; su uso habitual es como operando principal en operaciones aritméticas y lógicas, como base en accesos a memoria (LD, ST) y como primer entrada en comparaciones. La interpretación concreta depende del opcode (p. ej. en LD Rd, [RS1 + IMM32], RS1 es la base).

• **RS2** (Registro Fuente 2 - 4 bits): Segundo operando fuente codificado en 4 bits; sirve como operando secundario en instrucciones binarias, como índice en modos indexados ([RS1 + RS2]) cuando el opcode flags lo

permiten, o se rellena a cero cuando no es necesario. Su semántica es condicional al opcode y a los bits de función.

• **FUNC12** (Campo de función / modificador - 12 bits): Campo reservado para subfunciones y parámetros adicionales; su presencia evita consumir opcodes separados para variantes cercanas y añade flexibilidad sin fragmentar el espacio de opcode. FUNC12 se usa para:

- sub-opcodes o variantes internas de una familia.
- Cantidad de desplazamiento (shamt) en operaciones de shift.
- Indicadores de modo (por ejemplo sign-extend vs zero-extend en cargas de byte/half).
- Flags que activan modos alternativos (por ejemplo uso de RS2 como índice).

• Propuesta de distribución interna (suficiente y compacta):

i. bits [11:8] = sub-opcode / variante (4 bits).

ii. bits [7] = Flag de addressing indexado (si = 1 usar RS2 como índice y tratar IMM32 como offset).

iii. bits [6] = Flag de extensión en cargas (0 = zero-extend, 1 = sign-extend).

iv. bits [5:0] = shamt (6 bits, 0..63 para desplazar sobre 64 bits).

• **IMM32** (Campo inmediato/dirección - 32 bits): Campo dedicado a inmediatos aritméticos, offsets de memoria o direcciones absolutas en el espacio de 32 bits. Cuando IMM32 se interpreta como inmídiato aritmético se debe sign-extend a 64 bits antes de la operación; cuando se emplea como dirección o desplazamiento se trate como valor de 32 bits (Unsigned para direcciones absolutas, signed si el ensamblador lo interpreta como offset relativo). Para constantes de 64 bits se usan pseudo-instrucciones que el ensamblador descompone en secuencias reales.

Codificación de registros

Los registros de propósito general se codifican en 4 bits según la siguiente convención:

- R0 → 0000 • R2 → 0010 • R4 → 0100 • R6 → 0110 • R8 → 1000 • R10 → 1010
- R1 → 0001 • R3 → 0011 • R5 → 0101 • R7 → 0111 • R9 → 1001 • R11 → 1011

$$\bullet R12 \rightarrow 1100 \quad \bullet R13 \rightarrow 1101 \quad \bullet R14 \rightarrow 1110 \quad \bullet R15 \rightarrow 1111$$

Observación: R14 y R15 se reservan por convención para enlace y pila, respectivamente; R0 no se define obligatoriamente como registro constante cero para mantener generalidad funcional.

Tipos de Instrucción

- **R-type (Registro):** usa OPCODE, RD, RSI, RS2, FUNC12 (IMM32 = 0 normalmente). Operaciones binarias / Una salida.
- **I-type (Inmediato/memoria):** usa OPCODE, RD, RSI, FUNC12 opcional, IMM32 (inmediato o offset).
- **J-type (salto/llamada):** usa OPCODE, IMM (dirección absoluta o calculada); RD/RS ignorados o usados para enlaces (CALL).
- **S-type (sistema/efecto):** instrucciones de control y mantenimiento; campos OPCODE y algunos bits de FUNC12/IMM32 pueden contener parámetros; RD/RS frecuentemente 0.
- **Pseudo - Instrucción:** no mapea a una sola palabra real; el ensamblador la expande en una secuencia de instrucciones de los tipos anteriores.

Conjunto de instrucciones

1. Nombre: ADD

OPCODE (hex): 0x10

Tipo: R-type

Sintaxis: ADD RD, RSI, RS2

Descripción: Suma el valor de dos registros y guarda el resultado en el registro destino.

Semántica: $RD \leftarrow RSI + RS2$ (64-bit integer).

Campos significativos: OPCODE, RD, RSI, RS2, FUNC12=0.

Efectos sobre Flags: Z, N, C, V.

Notas: Overflow y Carry según aritmética.

2. Nombre: SUB

OPCODE: 0x11

Tipo: R-type

Sintaxis: SUB RD, RSI, RS2

Descripción: Resta el segundo operando del primero y escribe el resultado en el registro destino.

Semántica: $RD \leftarrow RSI - RS2$

Campos significativos: OPCODE, RD, RSI, RS2.

Flags afectadas: Z, N, C, V.

Notas: En caso de underFlow, se notifica al usuario al respecto.

3. Nombre: MUL

OPCODE: 0x12

Tipo: R-type

Sintaxis: MUL RD, RSI, RS2

Descripción: Multiplica los valores de dos registros y guarda la parte baja del producto en el destino.

Semántica: $RD \leftarrow (RS1 \times RS2) \bmod 2^{64}$

Campos significativos: OPCODE, RD, RSI, RS2.

Flags afectadas: Z, N (C/V por convención = 0).

Notas: truncamiento en overflow.

5. Nombre: AND

OPCODE: 0x14

Tipo: R-type

Sintaxis: AND RD, RSI, RS2

Descripción: Realiza la operación AND bit a bit entre dos registros y guarda el resultado.

Semántica: $RD \leftarrow RS1 \& RS2$

Campos significativos: OPCODE, RD, RSI, RS2.

Flags afectadas: Z, N.

4. Nombre: DIV

OPCODE: 0x13

Tipo: R-type

Sintaxis: DIV RD, RSI, RS2

Descripción: Divide el primer operando por el segundo y almacena el cociente en el destino.

Semántica: $RD \leftarrow \text{Floor}(RS1 / RS2)$ (entero)

Campos significativos: OPCODE, RD, RSI, RS2.

Flags afectadas: Z, N.

Notas: genera TRAP si RS2 == 0.

6. Nombre: OR

OPCODE: 0x15

Tipo: R-type

Sintaxis: OR RD, RSI, RS2

Descripción: Realiza la operación OR bit a bit entre dos registros y guarda el resultado.

Semántica: $RD \leftarrow RS1 | RS2$

Campos significativos: OPCODE, RD, RSI, RS2.

Flags afectadas: Z, N.

7. Nombre: XOR

OPCODE: 0x16

Tipo: R-type

Sintaxis: XOR RD, RSI, RS2

Descripción: Realiza la operación XOR bit a bit entre dos registros y guarda el resultado.

Semántica: $RD \leftarrow RSI \wedge RS2$

Campos significativos: OPCODE, RD, RSI, RS2

Flags afectadas: Z, N.

8. Nombre: NOT

OPCODE: 0x17

Tipo: R-type

Sintaxis: NOT RD, RSI

Descripción: Invierte todos los bits del registro Fuente y escribe el resultado en el destino.

Semántica: $RD \leftarrow \neg RSI$

Campos significativos: OPCODE, RD, RSI (RS2=0).

Flags afectadas: Z, N.

9. Nombre: SHL

OPCODE: 0x18

Tipo: R-type

Sintaxis: SHL RD, RSI, #shamt

Descripción: Desplaza lógicamente a la izquierda el valor de RSI por shamt posiciones y guarda en RD.

Semántica: $RD \leftarrow RSI \ll shamt$ (logical)

Campos significativos: OPCODE, RD, RSI, FUNC12

Flags afectadas: Z, N, C (bit shift-out)

Notas: Shamt 0..63.

10. Nombre: SHR

OPCODE: 0x19

Tipo: R-type

Sintaxis: SHR RD, RSI, #shamt

Descripción: Desplaza lógicamente a la derecha el valor de RSI por shamt posiciones y guarda en RD.

Semántica: $RD \leftarrow \text{logical-right}(RSI) \gg shamt$

Campos significativos: OPCODE, RD, RSI, FUNC12 (shamt).

Flags afectadas: Z, N, C.

II. Nombre: SAR

OPCODE: 0x1A

Tipo: R-type

Sintaxis: SAR RD, RSI, #shamt

Descripción: Desplaza aritméticamente a la derecha el valor de RSI preservando el signo y guarda en RD.

Semántica: $RD \leftarrow \text{arithmetic-right}(RSI) \gg shamt$

Campos significativos: OPCODE, RD, RSI, FUNC12 (shamt)

Flags afectadas: Z, N, C

12. Nombre: ADDI

OPCODE: 0x20

Tipo: I-type

Sintaxis: ADDI RD, RSI, IMM32

Descripción: Suma un inmediato sign-extended al contenido de un registro y guarda el resultado.

Semántica: $RD \leftarrow RSI + \text{sign-extend}(IMM32)$

Campos significativos: OPCODE, RD, RSI, IMM32

Flags afectadas: Z, N, C, V

Notas: IMM32 sign-extend.

13. Nombre: SUBI

OPCODE: 0x21

Tipo: I-type

Sintaxis: SUBI RD, RSI, IMM32

Descripción: Resta un inmediato sign-extended del registro fuente y almacena el resultado.

Semántica: $RD \leftarrow RSI - \text{sign-extend}(IMM32)$

Campos significativos: OPCODE, RD, RSI, IMM32

Flags afectadas: Z, N, C, V

14. Nombre: MOVI

OPCODE: 0x22

Tipo: I-type

Sintaxis: MOVI RD, IMM32

Descripción: Carga un inmediato de 32 bits (sign-extended) en el registro destino.

Semántica: $RD \leftarrow \text{sign-extend}(IMM32)$

Campos significativos: OPCODE, RD, IMM32

Flags afectadas: Z, N

Notas: Usar LI para constantes 64-bit.

15. Nombre: LD

OPCODE: 0x23

Tipo: I-type

Sintaxis: LD RD, [RSI + IMM32]

Descripción: Carga una palabra de 64 bits desde memoria en el registro destino usando base+offset.

Semántica: $RD \leftarrow \text{MEM}[RSI + \text{sign-extend}(IMM32)]$ (64 bits).

Campos significativos: OPCODE, RD, RSI, IMM32, FUNC12 (Extensiones).

Notas: FUNC12.bit6 para sign/zero extend en cargas menores.

16. Nombre: ST

Tipo: I-type

OPCODE: 0x24

Sintaxis: ST RS2, [RSI + IMM32]

Semántica: $\text{MEM}[RSI + \text{sign-extend}(IMM32)] \leftarrow RS2$ (64 bits)

Descripción: Almacena la palabra de 64 bits del registro fuente en memoria en la dirección base+offset.

Campos: OPCODE, RSI, RS2, IMM32

17. Nombre: LDB

OPCODE: 0x25

Tipo: I-type

Sintaxis: LDB RD, [RSI, IMM32]

Descripción: Carga un byte desde memoria, lo extiende (sign/zero según FUNC12) y lo escribe en RD.

Semántica: $RD \leftarrow \text{extend}(\text{MEM}[RSI + sign-extend(IMM32)], \text{FUNC12}.bit6)$.

Campos significativos: OPCODE, RD, RSI, IMM32, FUNC12.bit6.

Flags afectadas: Z, N.

Notas: Extensión controlable.

18. Nombre: STB

OPCODE: 0x26

Tipo: I-type

Sintaxis: STB RS2, [RSI + IMM32]

Descripción: Almacena el byte menos significativo del registro fuente en la dirección indicada.

Semántica: $\text{MEM}[RSI + sign-extend(IMM32)] - \text{byte } 0 \leftarrow RS2[7:0]$

Campos significativos: OPCODE, RSI, RS2, IMM32.

19. Nombre: LDABS

OPCODE: 0x27

Tipo: I-type

Sintaxis: LDABS RD, [IMM32]

Descripción: Carga una palabra de 64 bits desde la dirección absoluta especificada por IMM32.

Semántica: $RD \leftarrow \text{MEM}[IMM32]$

Campos significativos: OPCODE, RD, IMM32

Nota: Alternativa a LD con R0 base.

20. Nombre: STABS

OPCODE: 0x28

Tipo: I-type

Sintaxis: STABS RSI, [IMM32]

Descripción: Escribe una palabra de 64 bits desde RSI en la dirección absoluta IMM32.

Semántica: $\text{MEM}[IMM32] \leftarrow RSI$

Campos significativos: OPCODE, RSI, IMM32

21. Nombre: CMP

Sintaxis: CMP RSI, RS2

OPCODE: 0x30

Descripción: Compara dos registros actualizando las banderas, sin escribir resultado en registro.

Tipo: A-type

Semántica: $\text{FLAGS} = \text{Result_Flags}(RS1 - RS2)$

Campos significativos: OPCODE, RSI, RS2

Flags afectadas: Z, N, C, V

22. Nombre: CMPI

OPCODE: 0x31

Tipo: I-type

Sintaxis: CMPI RSI, IMM32

Descripción: Compara un registro con un inmediato y actualiza las banderas.

Semántica: FLAGS \leftarrow result_flags(RSI) - sign-extend(IMM32)).

Campos significativos: OPCODE, RSI, IMM32

Flags afectadas: Z, N, C, V

23. Nombre: TEST

OPCODE: 0x32

Tipo: I-type

Sintaxis: TEST RSI, RS2

Descripción: Realiza AND entre dos registros para probar bits y actualiza Z/N sin modificar registros.

Semántica: tmp \leftarrow RSI & RS2; FLAGS.Z/N \leftarrow Flags(tmp).

Campos significativos: OPCODE, RSI, RS2

Notas: Usada para máscaras.

24. Nombre: RD FLAGS

OPCODE: 0x33

Tipo: I-type

Sintaxis: RD FLAGS RD

Descripción: Copia el contenido del registro de banderas al registro destino en formato entero.

Semántica: RD \leftarrow zero-extend(FLAGS) (bits en LSB)

Campos significativos: OPCODE, RD

Notas: Lectura no privilegiada.

25. Nombre: WR FLAGS

OPCODE: 0x34

Tipo: I-type (Priv.)

Sintaxis: WR FLAGS RS

Descripción: Escribe el registro de banderas con los bits ofrecidos en RS (operación privilegiada).

Semántica: FLAGS \leftarrow RS [low bits]

Campos significativos: OPCODE, RSI

Flags afectadas: Todas las FLAGS.

Notas: Instrucción privilegiada.

26. Nombre: JMP

OPCODE: 0x40

Tipo: I-type

Sintaxis: JMP IMM32

Descripción: Cambia el contador de programa a la dirección absoluta indicada por IMM32.

Semántica: PC \leftarrow IMM32

Campos significativos: OPCODE, IMM32

Notas: Salto absoluto

27. Nombre: JZ

OPCODE: 0x41

Tipo: J-type

Sintaxis: JZ IMM32

Descripción: Si la bandera Zero está activa, salta a la dirección indicada; de lo contrario continua.

Semántica: if $Z == 1$ then $PC \leftarrow IMM32$
else $PC \leftarrow PC + 8$ bytes

Campos significativos: OPCODE, IMM32

28. Nombre: JNZ

OPCODE: 0x42

Tipo: J-type

Sintaxis: JNZ IMM32

Descripción: Salta a la dirección indicada si Zero está desactivada.

Semántica: if $Z == 0$ then $PC \leftarrow IMM32$
else $PC \leftarrow PC + 8$

Campos significativos: OPCODE, IMM32

29. Nombre: JC

OPCODE: 0x43

Tipo: J-type

Sintaxis: JC IMM32

Descripción: Salta a la dirección indicada si la bandera Carry está activada.

Semántica: if $C == 1$ then $PC \leftarrow IMM32$
else $PC \leftarrow PC + 8$

Campos significativos: OPCODE, IMM32

30. Nombre: JNC

OPCODE: 0x44

Tipo: J-type

Sintaxis: JNC IMM32

Descripción: Salta si la bandera Carry está desactivada.

Semántica: if $C == 0$ then $PC \leftarrow IMM32$
else $PC \leftarrow PC + 8$

Campos significativos: OPCODE, IMM32

31. Nombre: JS

OPCODE: 0x45

Tipo: J-type

Sintaxis: JS IMM32

Descripción: Salta si la bandera Negative indica número negativo.

Campos significativos: OPCODE, IMM32

32. Nombre: CALL

OPCODE: 0x46

Tipo: J-type

Sintaxis: CALL IMM32

Descripción: Guarda la dirección de retorno en R14 y salta a la subrutina indicada.

Campos significativos: OPCODE, IMM32

Notas: Uso de R14 como enlace.

33. Nombre: RET

OPCODE: 0x4F

Tipo: S-type

Sintaxis: RET

Descripción: Retorna de subrutina cargando el contador del programa desde R14.

Semántica: PC \leftarrow R14

Campos significativos: OPCODE

34. Nombre: BA

OPCODE: 0x48

Tipo: R-type

Sintaxis: BB RSI

Descripción: Realiza salto indirecto utilizando la dirección almacenada en un registro.

Semántica: PC \leftarrow RSI

Campos significativos: OPCODE, RSI

35. Nombre: PUSH

OPCODE: 0x50

Tipo: S-type

Sintaxis: PUSH RSI

Descripción: Decrementa el puntero de pila y salva la palabra de 64 bits del registro en memoria.

Semántica: SP \leftarrow SP-8; MEM[SP] \leftarrow RSI

Campos significativos: OPCODE, RSI

Nota: Pila crece hacia direcciones menores.

36. Nombre: POP

OPCODE: 0x51

Tipo: S-type

Sintaxis: POP RD

Descripción: Recupera la palabra superior de la pila en el registro destino y ajusta el puntero de pila.

Semántica: RD \leftarrow MEM[SP]; SP \leftarrow SP+8

Campos significativos: OPCODE, RD

Notas: Comprobar underflow por convención.

37. Nombre: ENTER

OPCODE: 0x52

Tipo: S-type

Sintaxis: ENTER IMM32

Descripción: Reserva en la pila un marco de tamaño IMM32 (decrementa SP).

Semántica: SP \leftarrow SP - IMM32

Campos significativos: OPCODE, IMM32

Notas: IMM32 múltiplo de 8 recomendado.

38. Nombre: LEAVE

OPCODE: 0x53

Tipo: S-type

Sintaxis: LEAVE IMM32

Descripción: Libera el marco de pila sumando IMM32 al puntero de pila.

Semántica: $SP \leftarrow SP + IMM32$

Campos significativos: OPCODE, IMM32

40. Nombre: OUT

OPCODE: 0x61

Tipo: I-type

Sintaxis: OUT RSI, IMM32

Descripción: Escribe el valor del registro en una dirección MMIO o puerto.

Semántica: $MMIO[IMM32] \leftarrow RSI$

Campos significativos: OPCODE, RSI, IMM32.

Nota: Posible bloqueo o handshake con dispositivo.

42. Nombre: INT_ENABLE

OPCODE: 0x63

Tipo: S-type

Sintaxis: INT_ENABLE

Descripción: Habilita el procesamiento de interrupciones externas marcando la bandera I.

Semántica: $FLAGS.I \leftarrow 1$

Campos significativos: OPCODE

Nota: Instrucción privilegiada.

39. Nombre: IN

OPCODE: 0x60

Tipo: I-type

Sintaxis: IN RD, IMM32

Descripción: Lee desde una dirección MMIO o puerto y coloca el dato en el registro destino.

Semántica: $RD \leftarrow MMIO[IMM32]$

Campos significativos: OPCODE, RD, IMM32

Notas: Sincronización dependiente del dispositivo.

41. Nombre: TRAP

OPCODE: 0x62

Tipo: I-type/S-type

Sintaxis: TRAP IMM8 (IMM8 en IMM32 LSB)

Descripción: Invoca un servicio de sistema o excepción software pasando el número de servicio.

Semántica: generar excepción/trap con vector = IMM8; guardar contexto según convención.

Campos significativos: OPCODE, IMM32 (IMM8), FUNC12 opcional.

Flags afectados: posible modificación por handler.

38. Nombre: TSTL

OPCODE: 0x64

Tipo: I-type

Sintaxis: TSTL RD, IMM32

Descripción: Compara el contenido de RD con la memoria en la dirección IMM32.

Campos significativos: OPCODE

Nota: Instrucción privilegiada.

43. Nombre: INT_DISABLE

OPCODE: 0x84

Tipo: S-type

Sintaxis: INT_DISABLE

Descripción: Deshabilita interrupciones externas limpando la bandera I.

Semántica: FLAGS.I < 0

Palabras Significativas: OPCODE

W: 1 byte

00X0: 300040

SINTAXIS: 10001000T

44. Nombre: IRET

OPCODE: 0x65

Tipo: S-type

Sintaxis: IRET

Descripción: Finaliza el servicio de interrupción restaurando el contexto (FLAGS y PC) desde la pila o registro.

Semántica: Restaurar FLAGS y PC según convención ISR.

45. Nombre: NOP

OPCODE: 0x70

Tipo: S-type

Sintaxis: NOP

Descripción: No realiza operación y avanza al siguiente PC; se usa para burbujas o alineamiento.

Semántica: PC ← PC + 8

Flags afectadas: Ninguna

W: 1 byte

00X0: 300040

SINTAXIS: 10001000T

46. Nombre: HALT

OPCODE: 0x71

Tipo: S-type

Sintaxis: HALT

Descripción: Detiene la ejecución del procesador hasta reinicio o intervención externa.

Semántica: Detener CPU (Estado HALT)

47. Nombre: CACHEFLUSH

OPCODE: 0x72

Tipo: S-type

Sintaxis: CACHEFLUSH

Descripción: Vacía completamente el contenido de la memoria cache, escribiendo de vuelta a memoria principal todas las líneas modificadas y marcando el resto como invariables.

Semántica: For line M Cache[l line, dirty == 1] then {MEM[l line, address] ← l line, data} endif l line.valid < 0 forend

48. Nombre: MEMBARRIER

OPCODE: 0x73

Tipo: S-type

Sintaxis: MEMBARRIER

Descripción: Garantiza el orden de las operaciones de memoria (fence) entre instrucciones anteriores y posteriores.

Semántica: Completar accesos de memoria según política de coherencia.

49. Nombre: LI (Pseudo)

OPCODE: 0x80 (Pseudo)

Tipo: Pseudo-instr

Sintaxis: LI RD, IMM64

Descripción: Carga una constante de 64 bits en el registro destino expandiendo la instrucción en varias reales.

Semántica: MOVI RD, IMM[31:0];
MOVI RT, IMM[63:32]; SHL RT, RT, #32;
OR RD, RD, RT

Notas: El registro R13 se reserva como registro temporal (RT) para esta pseudo-instrucción y no debe ser utilizado en código general.

50. Nombre: CRC

OPCODE: 0x90

Tipo: R-type

Sintaxis: CRC RD, RSI, RSZ

Descripción: Calcula un CRC/chequeo sobre un bloque o parámetros y almacena el resultado en RD.

Semántica: RD \leftarrow CRC 64 (parameters)

Campos significativos: OPCODE, RD, RSI, RSZ, FUNC12 opcional.

Ejemplo de uso:

Queremos calcular la suma de dos números almacenados en memoria y guardar el resultado en otro registro.

Secuencia en ensamblador:

LD R1, [R0 + 0x00000010]; Cargar operando A desde memoria

LD R2, [R0 + 0x00000018]; Cargar operando B desde memoria

ADD R3, R1, R2 ; Sumar A+B \rightarrow R3

ST R3, [R0 + 0x00000020]; Guardar el resultado en memoria

Resultado final (programa en binario, palabra a palabra):

1. 0x2310 0000 0000 0010

2. 0x2320 0000 0000 0018

3. 0x1031 2000 0000 0000

4. 0x2400 3000 0000 0020

El Cargador es la parte del enlazador-cargador que coloca un programa que ya está en código máquina (binario) en la memoria RAM para que pueda ejecutarse.

Nuestro cargador se encargará de:

1. Recibir el programa en código máquina.
2. Tomar una dirección inicial de carga.
3. Copiar el programa a la memoria RAM.
4. Configurar el (PC) program counter
5. Transferir control al CPU.

Lo primero que hay que tener en cuenta es que, según nuestro diseño la memoria RAM contiene las direcciones utilizables desde la $0x10000000$ - $0xFFFFFFF$ y nuestra instrucción consta de 64 bits, es decir, una palabra. Por lo anterior sabemos que el PC (program counter) va a ir saltando de la siguiente forma.

↓

$PC \leftarrow PC + 8$. Teniendo esto en cuenta, podemos empezar con el cargador recibiendo por parte del linker el programa en código máquina, (se dará un ejemplo más adelante) y revisa si cuenta con una dirección base, a partir de la cual empezará a cargar el programa, esta información se la va a pasar al PC y el va a iniciar desde esta a correr el programa.

Después de estos dos primeros pasos, va a empezar a ubicar el programa en las direcciones que encuentre disponibles en la memoria RAM y configura el PC para que inicie en la dirección base brindada (ej: en la dirección $0x10000000$), luego para transferir el control a la CPU como último paso ejecuta una instrucción de salto hacia la dirección base y la CPU empieza a ejecutar el programa.

Cargador dinámico: En caso tal, donde al cargador no se le brinde una dirección base, este se encargará de:

1. analizar la memoria RAM, ubicando las direcciones que están ocupadas y buscando un bloque de memoria contiguo suficientemente grande para ubicar el programa.
2. elegir la dirección base de ese bloque libre, tomando la primera dirección, como ejemplo, $0x10000000$.

3. Reubica direcciones, sumando la base a cualquier salto que tenga el programa, por ejemplo si se escoge la dirección base 0x1000008 y tiene un salto, lo calcula con el salto original, suponiendo que era a 0x1000832 salta a 0x1000040

Ejemplo : (basado en el ejemplo de aplicación de las instrucciones)

Se tienen las siguientes instrucciones en un código maquina.

Las cuales llamaremos "instrucción, i" durante la explicación para facilitar su comprensión.

Luego, se decide que la dirección base será 0x10000000 en la memoria ram, y empieza a cargar las instrucciones de la siguiente forma.

RAM [0x10000000] ← Instrucción-1
RAM [0x10000008] ← Instrucción-2
RAM [0x10000010] ← Instrucción-3
RAM [0x10000018] ← Instrucción-4

Luego le da al PC la dirección 0×10000000 que es la cual va a usar para empezar a leer el programa que se cargó.

Y comienza a ejecutarlo.

Algoritmo: Bubble sort

El bubble sort (o método de la burbuja) es un algoritmo de ordenamiento que itera repetidamente sobre una lista, comparando cada pareja de elementos adyacentes y cambiandolos si están en el orden incorrecto. Las pasadas a través de la lista se repiten hasta que no se necesiten más intercambios, lo que indica que la lista está ordenada. Es un algoritmo simple, pero inefficiente para grandes listas.

- Pseudocódigo

```
fun bubble-sort (arr : arreglo)
    n := longitud (arr)

    mientras (se-hicieron-swaps) hacer
        se-hicieron-swaps := falso

        para i desde 0 hasta n-2 hacer
            si arr[i] > arr[i+1] entonces
                intercambiar arr[i] y arr[i+1]
                se-hicieron-cambios := verdadero
            fsi
        fpara
        n := n - 1
    fmientras
ffon
```

- Código en ensamblador

Este código asume que un arreglo de 20 elementos de 64 bits está ubicado en la dirección de memoria 0x4000

Etiqueta	Instrucción
	MOVI R1, 0x4000
	MOVI R2, 20
	MOVI R3, 0x0
<u>outer-loop:</u>	MOVI R4, R2
	SUBI R2, R2, 1
	CMPI R2, 0x0
	JZ <u>sort_end</u>
	MOVI RS, R1

inner-loop:

```

SUBI R4, R4, 1
CMPI R4, 0x0
JZ check_swap_flag
LD R6, [RS + 0x0]
LD R7, [RS + 0x8]
CMP R7, R6
JS swap_elements
JMP next-pair

```

swap-elements:

```

ST R6, [RS + 0x8]
ST R7, [RS + 0x0]
MOVI R3, 1

```

next-pair:

```

ADDI RS, RS, 8
JMP inner-loop

```

check_swap_flag:

```

CMPI R3, 0x0
JZ sort-end
MOVI R3, 0
JMP outer-loop

```

sort-end:

```

HALT

```

- Traducción al lenguaje de máquina

Para la traducción se asume que las instrucciones se cargan en la memoria a partir de la dirección 16384 (0x4000)

Dirección	Instrucción máquina (hexadecimial)
16384	0x22100000000000004000
16392	0x222000000000000014
16400	0x223000000000000000
16408	0x224020000000000000
16416	0x21220000000000001
16424	0x310200000000000000
16432	0x4100000000000000110
16440	0x225010000000000000
16448	0x214400000000000001
16456	0x310400000000000000
16464	0x4100000000000000A8
16472	0x236500000000000000
16480	0x237500000000000008
16488	0x300706000000000000

16496	0x470000000000000080
16504	0x400000000000000098
16512	0x24650000000000008
16520	0x24750000000000000
16528	0x22300000000000001
16536	0x20550000000000008
16544	0x400000000000000040
16552	0x31030000000000000
16560	0x41000000000000110
16568	0x22300000000000000
16576	0x400000000000000018
16584	0x710000000000000000

- Verificación de la máquina

La implementación del Bubble Sort confirma que la máquina es capaz de soluciones a problemas relativamente complejos:

1. Manipulación de datos y memoria: La máquina puede cargar y almacenar datos de 64 bits (LD, ST) en la memoria. Esto es esencial para trabajar con arreglos.
2. Aritmética e incrementos de puntero: Las instrucciones como ADDI permiten sumar inmediatos a los registros para manipular punteros de memoria, lo cual es crucial para recorrer arreglos.
3. Lógica condicional y bucle anidado: El uso de CMP, JS, JMP permite construir bucles anidados y tomar decisiones basadas en comparaciones.
4. Manejo de banderas: La implementación de una "bandera" para los swaps demuestra la capacidad de la máquina para manejar variables de control que afectan el flujo del programa.

Algoritmos implementados:

Para realizar la prueba del funcionamiento de la computadora diseñada en base a las definiciones y consideraciones de hardware y lenguaje ensamblador realizadas, se desarrollaron tres algoritmos comunes de complejidad media en pseudocódigo y luego traducidos a lenguaje ensamblador para esta arquitectura siguiendo las instrucciones elementales definidas. Finalmente, se realizó la traducción de las instrucciones de los algoritmos a lenguaje de máquina especificando la dirección a partir de la cual se encuentran almacenadas.

Algoritmo de búsqueda lineal:

Es un algoritmo diseñado para encontrar un elemento en una colección de elementos o estructura de datos lineal, como un arreglo. Se compara cada elemento de una lista secuencialmente hasta encontrar el elemento buscado o llegar al final de la lista.

```

fun busqueda-lineal (arr: arreglo, elemento)
    n := longitud (arr)
    para i desde 0 hasta n-1 hacer
        si arr[i] == elemento
            retornar i
        fin si
    fin para
    retornar -1 // no encontrado
ffun

```

Nota: Si no encuentra el elemento en la estructura retorna -1.

Código ensamblador:

Se asume que el arreglo contiene elementos de 64 bits y se ubican a partir de la dirección 0x4000. n: tamaño del arreglo

Etiqueta

labeled

n = 10

elemento = 9

MOVI R5, 0x4000

MOVI R2, 10

MOVI R4, 0x0

MOVI R3, elemento ; elemento = 9

etiqueta

instrucción

loop:

CMP R4, R2

JN NO-ENCONTRADO

LD R6, [R5+0x0]

CMP R6, R3

JE ENCONTRADO

ADDI R4, R4, 1

ADDI R5, R5, 8

JMP LOOP

; 8 bytes por palabra

ENCONTRADO:

MOVI R14, 0

ADD R14, R14, R4

RET

NO ENCONTRADO:

MOVI R1, -1

RET

Código de máquina:

Se supone que la dirección estática donde se carga el programa es
24576 (0x6000)

dirección

Instrucción máquina (hexadecimal)

24576

0x2250000000004000

24584

0x222000000000000A

24592

0x2240000000000000

24600

0x2230000000000009

24608

0x3042000000000000

24616

0x4200000000006078

24624

0x2365000000000000

24632

0x3063000000000000

24640

0x4100000000006068

24648

0x2044000000000001

24656

0x2055000000000009

24664

0x4000000000006408

24672

0x22E0000000000000

24680

0x106E400000000000

24688

0x4700000000000000

24696

0x221100000000000D

24704

0x4700000000000000

Algoritmo de ordenamiento por inserción:

Es un algoritmo que funciona bien para listas pequeñas o parcialmente ordenadas. Ordena los elementos tomando uno de ellos y buscando la posición indicada para insertarlo.

```

fun ordenamientoInsercion (arr: arreglo, n)
para i desde 1 hasta n-1 hacer
    clave = arr[i]
    j = i - 1
    mientras j >= 0 ∧ arr[j] > clave hacer
        arr[j+1] = arr[j]
        j = j - 1
    fin-mientras
    arr[j+1] = clave
fin-par
ffun

```

Código ensamblador.

Se supone que el arreglo dado como argumento tiene elementos de 64 bits, donde n es la longitud del arreglo.

• Dirección base arreglo arr: 0x4000

Etiqueta

Instrucción

```

MOV I R1, 0x4000
MOVI R3, 1
MOVI R2, 10
CMP R3, R2
JE FIN-ORDENAMIENTO

```

BUCLE-EXTERNO:

```

MOVI R6, 0x0
ADD R6.B1, R6
MOVI R7, 8
MUL R7, R3, R7
ADD R6, R5, R7
LD R5, [R6 + 0x0]
SUBI R4, R3, 1

```

M A Scribe

Eiqueta

BUCLE-INTERNO:

Instrucción
CMP R4,DX0
JS INSERTAR

MOV R5,0
ADD R6,R1,R6
MOV R7,R6
MUL R7,R4,R7
ADD R6,R6,R7

LD R7,[R6+0x0]
CMP R7,R5
JS INSERTAR

ST R7,[R6+0x8]
JMP BUCLE-INTERNO

INSERTAR:

ADDI R4,R4,1
MOV R6,0
ADD R6,R1,R6
MOV R7,8
MUL R7,R4,R7
ADD R6,R6,R7
ST R5,[R6+0x0]

ADDI R3,R3,1
JMP BUCLE-EXTERNO

FIN-ORDENAMIENTO: HALT

Código máquina

Se supone que el código de máquina del programa se guarda
efectivamente a partir de la dirección 20480 (0x50000)

Dirección
20480
20489
20496
20504
20512
20520
20528
20536
20544
20552
20560
20568
20576

Instrucción máquina (hexadecimal)
0x22100000000000004000
0x22300000000000000001
0x2220000000b00000
0x30320000000000000000
0x41000000000005100
0x22600000000000000000
0x10616000000000000000
0x22700000000000000008
0x12737000000000000000
0x10667000000000000000
0x23560000000000000000
0x21431000000000000000
0x31400000000000000000

Dirección
 20584
 20592
 20600
 20C08
 20616
 20624
 20632
 20640
 20648
 20656
 20664
 20672
 20680
 20688
 20696
 20704
 20712
 20720
 20728
 20736

Instrucción máquina (hexadecimal)
 0x43400000000050B8
 0x2260000000000000
 0x1061600000000000
 0x2270000000000000
 0x1274700000000000
 0x1056700000000000
 0x2376000000000000
 0x3075000000000000
 0x45000000000050B8
 0x2476000000000008
 0x4000000000005060
 0x2044000000000001
 0x2260000000000000
 0x1044100000000000
 0x2278000000000000
 0x1274700000000000
 0x2456000000000000
 0x2033100000000000
 0x4000000000005060
 0x7100000000000000

- Verificación de la máquina:

Estos dos algoritmos que recorren estructuras lineales para buscar o crear nodos fueron compatibles en la implementación de la máquina diseñada, pagando por su proceso a lenguaje ensamblador con las instrucciones definidas, y luego a lenguaje de máquina expresado de forma hexadecimal en una dirección estática.

El diseño de las instrucciones elementales al ser atómicas permiten interactuar con el hardware, manejando accesos y escrituras sobre memoria pagando por registros.

Además, los saltos condicionales e incondicionales permiten dirigir el flujo en la lógica del programa, verificando los estados o flags de la máquina.