UNIVERSIDAD CENTRAL DE VENEZUELA

FACULTAD DE CIENCIAS

ESCUELA DE COMPUTACIÓN

Proyecto 1

Informe fase 1

Docente:
Ana Morales

Integrantes:
Diego León 31549189
Carlos Paccagnella 31752534
Aaron Contreras 31871275

Caracas, enero de 2026

Resumen

Para el desarrollo de la fase I de este proyecto, se implementó una arquitectura virtual en C que emula el funcionamiento de hardware real sobre el sistema operativo Linux. La solución se fundamenta en un diseño modular donde el archivo brain.h actúa como el núcleo de definiciones, centralizando las estructuras de datos, códigos de operación y constantes críticas del sistema. Este archivo contiene marcadores para facilitar la navegación.

Módulos del Sistema

Los componentes del hardware virtual se organizaron en los siguientes módulos analógicos:

- **cpu.h**: Implementa la lógica del procesador, el ciclo de instrucción y el manejo de registros.
- **memory.h**: Emula la memoria RAM de 2000 posiciones de 8 dígitos decimales cada una.
- **bus.h**: Gestiona el arbitraje y la comunicación entre dispositivos y memoria física.
- **dma.h**: Controla las transferencias de datos asíncronas entre el disco y la RAM.
- **disk.h**: Simula el almacenamiento masivo estructurado en pistas, cilindros y sectores.
- **load.h**: Responsable de la lectura y carga de archivos de programa en la memoria del sistema.
- **log.h**: Sistema de registro que documenta cronológicamente cada evento realizado por la arquitectura.

Esquema de Ejecución

1. El programa main.c inicia la interfaz de consola (shell) desde la cual el usuario puede cargar programas, ejecutarlos o activar el modo debugger para análisis detallado.
2. El loader interpreta los archivos de entrada y posiciona las instrucciones en la RAM basándose en las directivas del archivo.
3. La CPU ejecuta su ciclo de instrucción compuesto por las etapas de fetch, decode, execute, memory y write back, concluyendo con el chequeo de interrupciones.
4. Se utiliza un bus de sistema compartido que conecta la RAM con la CPU y el DMA, garantizando la integridad de los datos mediante un sistema de arbitraje.
5. El DMA emplea un hilo de trabajo independiente para ejecutar operaciones de entrada y salida de forma asíncrona, enviando una interrupción al procesador al finalizar la tarea.
6. El módulo de log mantiene un historial detallado de todas las acciones del sistema para verificar la correctitud de la arquitectura.

Planificación

Para el desarrollo del proyecto, se estableció una planificación estructurada en catorce tareas técnicas, siguiendo una progresión lógica desde la infraestructura base hasta la integración final:

1. **Infraestructura y Esqueleto**: Configuración del entorno de desarrollo, creación del Makefile y la estructura de archivos fuente del sistema.
2. **Definiciones del Sistema**: Especificación de constantes, registros, códigos de operación (opcodes) y estructuras de datos globales en brain.h.
3. **Gestión de Memoria y Bus**: Implementación de la memoria RAM y el bus de sistema con arbitraje mediante exclusión mutua (mutex).
4. **Sistema de Trazabilidad**: Desarrollo del módulo de log para el registro cronológico de eventos en archivo y consola.
5. **Cargador de Programas**: Implementación del loader para el procesamiento de archivos de texto y su escritura en la memoria física.
6. **Ciclo de Instrucción**: Programación de las etapas de fetch, decode y execute del procesador.
7. **Protección de Memoria**: Creación de la unidad de gestión de memoria (MMU) para la traducción de direcciones y validación de límites de segmentación.
8. **Instrucciones Base**: Implementación de la lógica aritmética, de transferencia de datos y comparación, manejando el formato de signo-magnitud.
9. **Interfaz de Usuario**: Desarrollo de la shell de comandos con soporte para ejecución normal y modo debug paso a paso.
10. **Subsistema de Almacenamiento y DMA**: Simulación del disco físico y creación de un hilo asíncrono para la gestión de entrada/salida mediante el DMA.
11. **Gestión de Interrupciones**: Implementación del mecanismo de salvaguarda de contexto en la pila y cambio a modo kernel ante excepciones o señales.
12. **Control de Flujo y Pila**: Programación de instrucciones de salto condicional y operaciones de manejo de la pila (PSH/POP).
13. **Pruebas e Integración**: Ejecución de casos de prueba integrales para validar la concurrencia entre CPU/DMA y redacción de la documentación técnica.
14. **Depuración de Hilos**: Verificación final del ciclo de vida de los hilos de ejecución para garantizar el cierre correcto del sistema.

Asunciones y Decisiones de Diseño

Durante la implementación de la arquitectura virtual, se tomaron decisiones clave respecto a la gestión de memoria y el comportamiento de los registros para garantizar la robustez y la seguridad del sistema. A continuación, se detallan las decisiones más relevantes:

- **Ubicación y Naturaleza de la Pila (Stack)**

  Se decidió ubicar la Pila del Sistema estrictamente dentro del Espacio Reservado del Sistema Operativo (direcciones físicas 0 a 299), aislandola completamente del espacio de memoria de usuario.

Dado que la función principal de la pila en esta arquitectura es salvaguardar el contexto crítico del procesador (`PC`, `AC`, `Flags`, etc.) durante las interrupciones y llamadas al sistema (`SVC`), es imperativo que estos datos residan en una zona de memoria protegida. Si la pila estuviera en el espacio de usuario, un proceso malintencionado o con errores podría sobrescribir el contexto guardado, provocando fallos catastróficos.

- **Estrategia de Crecimiento Descendente e Inicialización del SP**

El Registro Puntero de Pila (SP) se inicializa en la dirección 299, que corresponde a la última posición válida del espacio reservado del SO. Como consecuencia de esta lógica, la dirección física 299 nunca contiene datos de la pila, permaneciendo conceptualmente "vacía". Esta dirección actúa como base.

- **Vector de Interrupciones**

Aunque el espacio del sistema abarca desde la dirección 0 hasta la 299, se decidió establecer un límite de seguridad (hard-limit) para el crecimiento de la pila en la dirección 30, dado que el manejo de interrupciones se realiza mediante un vector indexado. En esta Implementación, este vector reside en las direcciones bajas de memoria (0-19), donde cada espacio de memoria apunta a una única dirección de memoria: la dirección física 20, En la dirección 20, se escribe manualmente la instrucción `RETRN` (Opcode 14), con la finalidad de simular el manejador de la interrupción, cabe destacar que el el manejador se ejecuta solo en interrupciones no fatales como realizar una SVC por ejemplo.

- **Tratamiento y Aislamiento de la Memoria de Usuario**

Se decidió que todos los programas de usuario se compilen y carguen pensando que su memoria comienza en la dirección 0, Se definió la constante `USER_PROGRAM_START` en 300. El *Loader* carga el binario físicamente a partir de esta dirección, y el registro Base (RB) se inicializa con este valor.

Brain

Constantes

Este bloque define constantes generales utilizadas en todo el sistema.

Incluye:

- MEM_SIZE: tamaño total de la memoria física del sistema.
- OS_RESERVED: cantidad de posiciones de memoria reservadas para el sistema operativo.
- WORD_DIGITS: número de dígitos decimales que conforman una palabra.

Estas constantes permiten estandarizar el tamaño de memoria y el formato de datos manejados por el simulador.

Modos de Operación

Este bloque define los modos en los que puede ejecutarse el procesador.

Incluye:

- USER_MODE: ejecución en modo usuario, con restricciones de memoria y privilegios.
- KERNEL_MODE: ejecución en modo kernel, con acceso completo al sistema.

Estos valores son utilizados por el PSW para controlar el nivel de privilegio durante la ejecución.

Códigos de Interrupción (Vector)

Este bloque define los códigos utilizados para indexar el vector de interrupciones.

Incluye interrupciones asociadas a:

- llamadas al sistema
- instrucciones inválidas
- direccionamiento inválido
- reloj

- finalización de operaciones de E/S
- overflow y underflow aritmético

Cada código identifica el tipo de evento que debe ser atendido por el manejador correspondiente.

Conjunto de Instrucciones (Opcodes)

Este bloque define los códigos de operación soportados por la arquitectura simulada.

Se agrupan en:

- Aritméticas: suma, resta, multiplicación y división.
- Transferencia de datos: carga y almacenamiento entre memoria y registros.
- Comparación y saltos: instrucciones de control de flujo condicional.
- Sistema: llamadas al sistema, retorno, habilitación de interrupciones y cambio de modo.
- Registros Base/Límite/Pila: manejo de protección de memoria y pila.
- E/S (DMA): configuración y activación del subsistema DMA.

Estos códigos son utilizados por el ciclo de ejecución del CPU para identificar la acción a realizar.

Estructuras de Datos

Este bloque agrupa las definiciones de tipos y estructuras fundamentales del sistema.

Definición de Palabra

Word representa una palabra de la arquitectura simulada.

- Se define como un entero en C.
- Modela palabras de 8 dígitos decimales utilizadas para instrucciones y datos.

Registro PSW (Program Status Word)

PSW_t representa el estado operativo del procesador.

Incluye:

- CC: código de condición.
- Mode: modo de ejecución (usuario o kernel).
- Interrupts: habilitación de interrupciones.
- PC: contador de programa.

Este registro controla la ejecución del CPU y la atención de interrupciones.

Contexto de Registros de CPU

CPU_Context representa el estado completo del procesador.

Incluye:

- Registros principales (AC, IR)
- Registros de acceso a memoria (MAR, MDR)
- Registros de protección (RB, RL)
- Registro índice (RX)
- Puntero de pila (SP)
- PSW asociado al contexto

Este contexto es compartido por los módulos del sistema y se actualiza durante la ejecución del ciclo de instrucción.

Estructura del DMA

DMA_t representa el estado interno del subsistema DMA.

Incluye:

- Parámetros de disco (TRACK, CYLINDER, SECTOR)
- Tipo de operación de E/S (IO)
- Dirección física de memoria (ADDRESS)
- Estado de la operación (STATE)
- Indicador de ocupación (BUSY)

- Mutex para sincronización concurrente (lock)

Esta estructura permite simular operaciones de entrada/salida concurrentes con el CPU.

DMA_BUSY_CODE define un valor especial utilizado para indicar que el DMA se encuentra ocupado cuando se solicita una nueva operación de E/S.

Main

Este módulo representa el punto de control del simulador. Su implementación coordina la inicialización de todos los subsistemas y provee una interfaz de consola para la interacción con el usuario.

void print_registers()

Parámetros: void

Descripción: Imprime por pantalla el estado actual de los registros del CPU usando el contexto global context.

Implementación:

1. Imprime un encabezado de estado.
2. Muestra PC, IR y AC desde context.
3. Muestra RX, SP y el modo (User/Kernel) según context.PSW.Mode.
4. Muestra RB, RL y el código de condición CC.
5. Imprime una línea para separar la salida.

Utiliza: none
Referenciada en: main

void init_kernel()

Parámetros: void

Descripción: Inicializa un vector básico de interrupciones en memoria para evitar fallos por direcciones inválidas.

Implementación:

1. Escribe el valor 20 en memoria física desde la dirección 0 hasta la 19, esto para simular que en la posición 20 existe el manejador de interrupciones.
2. Escribe 14000000 en la dirección 20 como RETRN (simulación del manejador).
3. Escribe 0 en memoria física desde la dirección 21 hasta la 29.
4. Registra en el log el mensaje de inicialización del vector.

Utiliza: mem_write_physical, write_log
Referenciada en: system_init

int system_init()

Parámetros: void

Descripción: Inicializa los módulos principales del sistema (bus, disco, DMA y CPU)

Implementación:

1. Registra en el log el inicio del sistema.
2. Inicializa el bus con bus_init, si falla retorna -1, error
3. Inicializa el disco con disk_init, si falla retorna -1, error
4. Inicializa el DMA con dma_init, si falla retorna -1, error
5. Reinicia los registros del CPU con cpu_init.
6. Inicializa la memoria del kernel con init_kernel.
7. Imprime en consola el mensaje de éxito.
8. Retorna 0 si todo fue exitoso.

Utiliza: write_log, bus_init, disk_init, dma_init, cpu_init, init_kernel
Referenciada en: main

static void print_banner()

Parámetros: void

Descripción: Muestra en consola el menú de comandos disponibles en la shell del simulador.

Implementación:

1. Imprime el título de la shell.
2. Imprime la lista de comandos soportados (cargar, ejecutar, debug, salir).

Utiliza: none
Referenciada en: main

void reset(loadParams *info)

Parámetros:

loadParams *info: estructura con metadatos del programa cargado (load_address, n_words, index_start).

Descripción: Verifica si el programa terminó y lo reinicia de ser necesario.

Implementación:

1. Verifica si el modo actual es User_mode y si context.PSW.PC >= info->n_words.
2. Si la condición se cumple, reinicia registros con cpu_init.
3. Restaura RB con info->load_address.
4. Restaura RL con 1999.
5. Restaura PC con info->index_start.
6. Restaura SP con SYSTEM_STACK_START.
7. Restaura Mode a USER_MODE.

Utiliza: cpu_init
Referenciada en: main

const char *get_mnemonic(int opcode)

Parámetros:

int opcode: código de operación numérico.

Descripción: Traduce un opcode a texto en el Debugger.

Implementación:

1. Evalúa el opcode usando switch.
2. Retorna el mnemónico correspondiente cuando el opcode está definido.
3. Retorna UNKNOWN cuando el opcode no coincide con ninguno de los casos.

Utiliza: none
Referenciada en: main

int main()

Parámetros: void

Descripción: Implementa el shell interactivo del simulador, permitiendo cargar programas, ejecutarlos en modo normal, depurarlos paso a paso y finalizar el sistema liberando recursos.

Implementación:

1. Declara cargado e instancia loadParams info.
2. Inicializa el log con log_init.
3. Inicializa el sistema con system_init, si falla retorna -1, error
4. Muestra el menú con print_banner.
5. Entra en un ciclo que lee comandos hasta que se introduzca el de Salida.
6. Comando salir:

   Imprime mensaje de apagado.

   Libera recursos con dma_destroy, disk_destroy, bus_destroy y log_close.

   Rompe el ciclo principal.
7. Comando cargar <archivo>:

   Extrae el nombre del archivo, omitiendo espacios iniciales.

   Elimina espacios finales del nombre.

   Llama a load_program(filename, USER_PROGRAM_START, &info).

   Si carga exitosamente, reinicia registros con cpu_init y configura RB, RL, PC, SP y Mode.

   Marca cargado = 1; si falla, marca cargado = 0.
8. Comando ejecutar:

   Verifica que exista programa cargado.

   Llama a reset(&info).

   Ejecuta ciclos de CPU llamando cpu() repetidamente.

   Detiene ejecución si cpu() retorna error.

   En modo usuario, detiene ejecución si PC >= info.n_words.

   Imprime registros con print_registers al finalizar.
9. Comando debug:

   Verifica que exista programa cargado.

   Llama a reset(&info).

Registra activación de debug en el log y muestra registros.

subcomandos:

step: calcula dirección física, lee instrucción con mem_read_physical, obtiene opcode, muestra mnemónico con get_mnemonic, ejecuta cpu() y muestra registros.

regs: imprime registros.

salir: registra desactivación y termina el bucle de depuración.

10. Si el comando no es reconocido y no está vacío, imprime mensaje de comando no reconocido.

11. Retorna 0 al finalizar.

Utiliza: log_init, system_init, print_banner, load_program, cpu_init, reset, cpu, print_registers, mem_read_physical, get_mnemonic, write_log, dma_destroy, disk_destroy, bus_destroy, log_close

Referenciada en: none

LOADER

int load_program(const char *filename, int base_address, loadParams *info)

Parámetros:

- const char *filename: nombre o ruta del archivo que será cargado en memoria.
- int base_address: dirección física base donde comenzará la carga del programa.
- loadParams *info: estructura donde se almacenan los datos del programa cargado.

Descripción: La función se encarga de cargar un programa desde un archivo de entrada hacia la memoria física del sistema.

Implementación:

1. Intenta abrir el archivo indicado por filename en modo lectura.
2. Si el archivo no puede abrirse, registra el error en el log y retorna -1.
3. Inicializa variables internas para el proceso de carga:

   Un contador offset para contabilizar cuántas palabras se han cargado realmente.

   Una variable declared_words para almacenar el valor declarado en NumeroPalabras (si existe).

4. Inicializa la estructura info:

   Guarda base_address como dirección de carga.

   Inicializa n_words en cero.

   Inicializa index_start en cero como valor por defecto.

5. Registra en el log el inicio del proceso de carga, indicando el archivo y la dirección física de carga.
6. Lee el archivo línea por línea.
7. En cada línea, intenta extraer el primer token:

   Si la línea está vacía o no contiene un token válido, la función la ignora.

8. Descarta líneas que sean comentarios completos.
9. Procesa directivas del archivo:

Si detecta _start, lee el número de línea indicado y ajusta el índice a base cero (lineaStart - 1 si es mayor que 0), guardándolo en info->index_start, y lo registra en el log.

Si detecta .NumeroPalabras, almacena el número declarado en declared_words y lo registra en el log.

Si detecta .NombreProg, lee el nombre del programa y lo registra en el log.

10. Si la línea contiene una palabra numérica:

    Convierte el contenido a Word.

    Calcula la dirección física destino

    Escribe la palabra en memoria física usando bus_write, indicando como identificador de cliente el valor 2 (Loader).

11. Si la escritura en memoria falla:

    Registra el error en el log.

    Cierra el archivo.

    Retorna -1.

12. Si la escritura es exitosa, incrementa offset para reflejar una palabra cargada adicional.
13. Al finalizar cierra el archivo.
14. Si se encontró NumeroPalabras, valida que:

    offset sea igual a declared_words.

    Si no coincide, la función registra la inconsistencia en el log y retorna -1.

15. Actualiza n_words con el valor real de palabras cargadas.
16. Registra en el log que la carga finalizó exitosamente e indica cuántas palabras fueron escritas.
17. Retorna 0 para indicar éxito.

Utiliza: bus_write, write_log
Referenciada en: main

LOG

Este módulo representa el sistema de registro de eventos del simulador. Su implementación se basa en un archivo de texto donde se almacenan mensajes generados por los distintos módulos del sistema. Para garantizar acceso seguro, utiliza un mutex que protege las operaciones de escritura. Su funcionamiento consiste en registrar eventos relevantes del CPU, la memoria, el bus, el DMA y el cargador, permitiendo el seguimiento de la ejecución y facilitando la depuración del sistema.

void log_init()

Parámetros: void

Descripción: Inicializa el sistema de logging del sistema

Implementación:

1. Inicializa el mutex log_lock mediante pthread_mutex_init.
2. Si ocurre un error durante la inicialización del mutex, se muestra el mensaje correspondiente y se termina la ejecución.
3. Abre el archivo log.txt en modo escritura.
4. Si el archivo no puede abrirse, se muestra el error.
5. Registra en el log un mensaje indicando el inicio del sistema.

Utiliza: write_log
Referenciado en: main

void log_close()

Parámetros: void

Descripción: Finaliza el sistema de logging, cerrando el archivo de registro.

Implementación:

1. Verifica si el descriptor log_file se encuentra inicializado.
2. Escribe un mensaje final en el archivo de log.
3. Cierra el archivo log.txt.
4. Libera el mutex utilizando pthread_mutex_destroy.

Utiliza: none
Referenciada en: main

void write_log(int console, const char *format, ...)

Parámetros:

- int console: indica si el mensaje también debe mostrarse en consola.
- const char *format: cadena de formato del mensaje.
- ...: argumentos variables asociados al formato.

Descripción: Registra mensajes en el archivo de log con marca de tiempo

Implementación:

1. Verifica si el archivo de log se encuentra abierto; en caso contrario, retorna.
2. Adquiere el mutex log_lock para garantizar acceso.
3. Obtiene la hora actual del sistema.
4. Convierte la hora a una cadena legible mediante strftime.
5. Escribe la marca de tiempo en el archivo de log.
6. Inicializa el manejo de argumentos variables con va_start.
7. Escribe el mensaje formateado en el archivo utilizando vfprintf.
8. Finaliza el manejo de argumentos variables con va_end.
9. Fuerza la escritura inmediata en el archivo con fflush.
10. Si console está habilitado, imprime el mismo mensaje en la salida estándar.
11. Libera el mutex.

Utiliza: none

Referenciada en: log_init, bus_init, bus_destroy, mem_read_physical, mem_write_physical, load_program, dma_init, dma_handler, dma_perform_io, dma_destroy, push_stack, pop_stack, get_value, mmu_translate, cpu_init, cpu_interrupt, handle_interrupt, cpu, main

BUS

Este módulo representa el bus del sistema encargado de intermediar el acceso a la memoria física. Su implementación utiliza un mecanismo de exclusión mutua para garantizar que solo un componente del sistema pueda acceder a la memoria a la vez. El bus recibe solicitudes de lectura y escritura tanto del CPU como del DMA, y su funcionamiento consiste en arbitrar estas solicitudes y redirigirlas al módulo de memoria de forma segura.

int bus_init()

Parámetros: Void

Descripción: Inicializa el bus del sistema

Implementación:

1. Inicializa la memoria del sistema llamando la función mem_init.
2. Inicializa el mecanismo de arbitraje del bus mediante la creación del mutex.
3. Si la inicialización del mutex falla se registra el error en el archivo de log llamando a write_log.
4. En caso de error durante la inicialización, la función retorna -1 para indicar fallo.
5. Si la inicialización es exitosa, se registra en el log que el bus fue inicializado correctamente.
6. Retorna 0 para indicar que la inicialización fue exitosa.

Utiliza: mem_init, write_log
Referenciada en: system_init

int bus_read(int address, Word *data, int client_id)

Parámetros:

- int address: dirección física de memoria desde la cual se desea realizar la lectura.
- Word *data: puntero donde se almacenará el dato leído desde memoria.
- int client_id: identificador del solicitante del acceso al bus (CPU o DMA).

Descripción: Realiza la lectura segura de una posición de memoria física

Implementación:

1. Solicita acceso al bus adquiriendo el mutex bus_lock.
2. Obtenido el acceso realiza la lectura física de memoria mediante la función mem_read_physical.
3. El valor leído es almacenado.
4. Finalizada la lectura, libera el bus liberando el mutex.
5. Retorna el resultado de la lectura.

Utiliza: mem_read_physical
Referenciada en: pop_stack, get_value, handle_interrupt, cpu, dma_perform_io

int bus_write(int address, Word data, int client_id)

Parámetros:

- int address: dirección física de memoria donde se realizará la escritura.
- Word data: dato que será escrito en la memoria.
- int client_id: identificador del solicitante del acceso al bus (CPU, DMA o Loader).

Descripción: Realiza la escritura segura en una posición de memoria física

Implementación:

1. Solicita acceso al bus adquiriendo el mutex bus_lock.
2. Obtenido el acceso, realiza la escritura física en memoria mediante la función mem_write_physical.
3. El resultado de la operación es almacenado.
4. Finalizada la escritura, libera el bus liberando el mutex
5. Retorna el resultado de la operación de escritura.

Utiliza:, mem_write_physical
Referenciada en: push_stack, cpu, dma_perform_io, load_program

void bus_destroy()

Parámetros: void

Descripción: Libera los recursos asociados al bus del sistema al finalizar la ejecución del programa.

Implementación:

1. Destruye el mutex utilizado para el arbitraje del acceso al bus.
2. Registra en el archivo de log que el bus ha sido finalizado correctamente.

Utiliza: write_log

Referenciada en: main

MEMORY

Este módulo representa la memoria principal del sistema. Se implementó como un arreglo lineal de palabras que modela la memoria RAM física del sistema. Consiste en almacenar instrucciones y datos, ofreciendo operaciones de lectura y escritura sobre direcciones físicas, y validando que los accesos se realicen dentro de los límites permitidos.

void mem_init()

Parámetros: Void

Descripción: Inicializa la memoria RAM física del sistema, limpiando dicha memoria.

Implementación:

1. Cada posición de memoria es inicializada con el valor cero utilizando memset

Utiliza: none
Referenciada en: bus_init

int mem_read_physical(int address, Word *value)

Parámetros:

- int address: dirección física de memoria desde la cual se desea realizar la lectura.
- Word *value: puntero donde se almacenará el dato leído desde memoria.

Descripción: Realiza la lectura de una posición de memoria física, validando previamente que la dirección solicitada se encuentre dentro de los límites.

Implementación:

1. Verifica que la dirección física recibida se encuentre dentro del rango válido de la memoria.
2. Si la dirección es menor que cero o mayor o igual al tamaño máximo de la memoria, retorna -1, un valor de error.
3. Si la dirección es válida, accede a la posición correspondiente del arreglo de memoria.
4. El valor leído es almacenado.
5. La operación de lectura es registrada en log, indicando la dirección accedida y el valor obtenido.
6. Retorna cero para indicar que la operación fue realizada exitosamente.

Utiliza: write_log
Referenciada en: bus_read, main

int mem_write_physical(int address, Word value)

Parámetros:

- int address: dirección física de memoria donde se desea realizar la escritura.
- Word value: dato que será almacenado en la memoria.

Descripción: Permite realizar la escritura directa de un dato en una posición de memoria física, asegurando que la dirección utilizada sea válida.

Implementación:

1. Verifica que la dirección física recibida se encuentre dentro del rango válido de la memoria.
2. Si la dirección es inválida, retorna -1, un valor de error.
3. Si la dirección es válida, escribe el valor recibido en la posición correspondiente del arreglo de memoria.
4. La operación de escritura es registrada en el log, indicando la dirección accedida y el valor escrito.
5. Retorna cero para indicar que la operación fue realizada exitosamente.

Utiliza: write_log
Referenciada en: bus_write, main

CPU

Este módulo representa el procesador del sistema. Su implementación modela el ciclo de instrucción completo, incluyendo búsqueda, decodificación y ejecución de instrucciones, manejo de interrupciones y control de modos de operación. Interpreta las instrucciones cargadas en memoria y coordina la interacción con la memoria, el bus y el DMA.

Funciones:

int push_stack(int value)

Parámetros:

int value: valor que será almacenado en la pila del sistema.

Descripción: Guarda un valor en la pila del sistema, tomando en cuenta condiciones de desbordamiento.

Implementación:

1. Decrementa el puntero de pila context.SP para reflejar el crecimiento descendente de la pila.
2. Verifica si el nuevo valor de SP invade el área reservada del sistema operativo (direcciones menores a 30).
3. En caso de desbordamiento:

   Registra el evento en el log.

   Restaura el valor original de SP.

   Retorna error.
4. Escribe el valor recibido en la dirección física indicada por SP utilizando bus_write.
5. Retorna cero, éxito si la escritura se realiza correctamente.

Utiliza: write_log, bus_write
Referenciada en: handle_interrupt, cpu

int pop_stack(int *value)

Parámetros:

int *value: puntero donde se almacenará el valor sacado de la pila.

Descripción: Saca un valor de la pila

Implementación:

1. Verifica si SP se encuentra en el límite superior permitido para la pila.
2. Si ocurre subdesbordamiento:

   Registra el evento en el log.

Retorna error.

3. Lee el valor almacenado en la dirección indicada por SP usando bus_read.

4. Incrementa SP para reflejar la liberación de la posición de pila.

5. Retorna cero, éxito si la lectura se realiza correctamente.

Utiliza: write_log, bus_read
Referenciada en: cpu

int get_value(int mode, int operand, int *value)

Parámetros:

- int mode: modo de direccionamiento.
- int operand: operando extraído de la instrucción.
- int *value: puntero donde se colocará el valor resultante.

Descripción: Obtiene el valor codificado en la instrucción según el modo de direccionamiento.

Implementación:

1. Evalúa si el modo de direccionamiento es 1. En tal caso copia directamente el operando en value y finaliza.

2. Verifica que el modo de direccionamiento sea válido.

3. Calcula la dirección lógica según el modo:

   Modo = 0: utiliza el operando.

   Modo = 2: suma el operando con el registro RX.

4. Traduce la dirección lógica a física mediante mmu_translate.

5. Si la traducción falla, retorna -1, error.

6. Lee el contenido de memoria física usando bus_read.

7. Copia el valor leído en value.

8. Retorna cero, éxito.

Utiliza: write_log, mmu_translate, bus_read
Referenciada en: cpu

int mmu_translate(int logical_addr)

Parámetros:

int logical_addr: dirección lógica generada por el CPU.

Descripción: Traduce las direcciones logicas del programa a fisicas para compatibilidad con la ram.

Implementación:

1. Verifica si el modo de ejecución es kernel.
2. En modo kernel, retorna la dirección sin modificación.
3. En modo usuario, suma la dirección lógica con el registro base RB.
4. Verifica que la dirección física resultante esté dentro del rango [RB, RL]. En caso de que ocurra esta violación de segmento:

   Registra el evento en el log.

   Solicita interrupción por dirección inválida.

   Retorna -1, error.
5. Retorna la dirección física válida.

Utiliza: write_log, cpu_interrupt
Referenciada en: get_value, cpu

void decode(Word instruction, int *opcode, int *mode, int *operand)

Parámetros:

- Word instruction: palabra de instrucción completa.
- int *opcode, int *mode, int *operand: campos resultantes.

Descripción:
Etapa decode del ciclo de instruccion del cpu, descompone una instrucción en sus campos correspondientes.

Implementación:

1. Copia la instrucción en una variable auxiliar.

2. Extrae el operando (últimos 5 digitos) utilizando módulo de 100000.
3. Elimina el operando de la instrucción dividiendo el auxiliar entre 100000.
4. Extrae el modo de direccionamiento con modulo de 10.
5. Elimina el modo de direccionamiento dividiendo auxiliar entre 10.
6. Asigna el resto (últimos dos dígitos) como opcode.

Utiliza: none
Referenciada en: cpu

void cpu_init()

Argumentos: void

Descripción: inicializa el procesador.

Implementación:

1. Inicializa en cero los registros.
2. Inicializa el puntero de pila en el límite superior 299.
3. Inicializa el PSW en modo usuario con interrupciones habilitadas.
4. Limpia cualquier interrupción pendiente.
5. Registra el evento de inicialización en el log.

Utiliza: write_log
Referenciada en: system_init

void cpu_interrupt(int interrupt_code)

Parámetros:

int interrupt_code: código de interrupción que se solicita marcar como pendiente.

Descripción: Marca una interrupción como pendiente para que sea atendida por el ciclo del CPU, almacenando el código de interrupción solicitado.

Implementación:

1. Activa la bandera interrupt_pending asignándole el valor 1.

2. Guarda el código recibido en la variable global interrupt_code_val.

3. Registra en el log la solicitud de interrupción, incluyendo el código recibido

4. Deja preparada la atención de interrupción para el siguiente ciclo de CPU, siempre que el PSW tenga interrupciones habilitadas.

Utiliza: write_log
Referenciada en: mmu_translate, cpu, dma_perform_io

int handle_interrupt()

Parámetros: void

Descripción: Maneja las interrupciones del cpu.

Implementación:

1. Registra en el log el inicio del proceso de atención de la interrupción pendiente, incluyendo el código de interrupción almacenado.
2. Verifica de que tipo de interrupción se trata, una vez identificado, se registra en el log el tipo de interrupción, se limpia la marca de interrupción pendiente y se retorna el valor que identifica la interrupción.
3. En caso de interrupción de E/S, si el estado indica que la operación de DMA no finalizó correctamente:

   Registra el error en el log.

   Limpia la marca de interrupción pendiente.

   Retorna un código de error asociado a fallo de E/S.

4. Inicia el proceso de salvado de contexto del proceso interrumpido.
5. Guarda el contador de programa (PC) en la pila del sistema utilizando push_stack.
6. Guarda el acumulador (AC) en la pila para preservar resultados parciales.
7. Guarda el registro índice (RX) en la pila.
8. Guarda el registro base (RB) en la pila.
9. Guarda el registro límite (RL) en la pila.
10. Guarda el código de condición (CC) del PSW en la pila.
11. Guarda el modo de ejecución actual (usuario o kernel) en la pila.
12. Cambia el modo de ejecución del PSW a KERNEL_MODE, garantizando privilegios completos durante la atención de la interrupción.

13. Deshabilita nuevas interrupciones colocando context PSW Interrupts en cero, evitando interrupciones anidadas no controladas.
14. Accede al vector de interrupciones para obtener la dirección del manejador correspondiente:

    Utiliza el código de interrupción como índice del vector.

    Lee la dirección física del manejador usando bus_read.

15. Carga la dirección del manejador en el contador de programa (context.PSW.PC), provocando el salto inmediato a la rutina de servicio.
16. Registra en el log el cambio de flujo de ejecución hacia el manejador de interrupción.
17. Limpia la marca de interrupción pendiente
18. Retorna 0, para indicar que el proceso de atención de interrupción se realizó correctamente.

Utiliza: write_log, dma_get_state, push_stack, bus_read
Referenciada en: cpu

int int_to_sm(int int_val)

Parámetros:

int int_val: entero en formato nativo de C que se desea convertir a signo–magnitud.

Descripción: Convierte un entero de C a representación signo–magnitud de 8 dígitos, aplicando control de rango de 7 dígitos para la magnitud y actualizando el código de condición ante overflow cuando corresponde.

Implementación:

1. Determina si el valor recibido es negativo:

   Si int_val < 0, selecciona signo negativo.

   Si int_val >= 0, selecciona signo positivo.

2. Convierte el valor a magnitud positiva:

   Si el signo es negativo, toma el valor absoluto como magnitud.

   Si el signo es positivo, usa el mismo valor como magnitud.

3. Verifica si la magnitud excede el límite permitido por la arquitectura (7 dígitos):

   El límite máximo permitido es 9999999.

4. Si la magnitud excede el límite:

   Registra en el log el evento de overflow.

   Ajusta la magnitud al valor máximo 9999999.

   Actualiza el código de condición context.PSW.CC con el valor 3, indicando desbordamiento.

5. Construye el valor signo–magnitud:

   Coloca el dígito de signo como el primer dígito (0 para positivo, 1 para negativo).

   Coloca la magnitud en los 7 dígitos restantes.

6. Retorna el resultado final en formato signo–magnitud.

Utiliza: write_log
Referenciada en: cpu

int cpu()

Parámetros: void

Descripción: Ejecuta un ciclo completo de instrucción.

Implementación:

1. Simula el retardo del ciclo de CPU mediante usleep.
2. Verifica si existe una interrupción pendiente y si las interrupciones están habilitadas.
3. Atiende la interrupción pendiente mediante handle_interrupt cuando corresponde.
4. Copia el contador de programa en MAR.
5. Traduce la dirección lógica a física mediante mmu_translate.
6. Lee la instrucción desde memoria usando bus_read.
7. Carga la instrucción en IR.
8. Incrementa el contador de programa.
9. Decodifica la instrucción usando decode.

10. Evalúa el opcode mediante una estructura switch.
11. Obtiene operandos usando get_value cuando la instrucción lo requiere.
12. Ejecuta operaciones aritméticas y lógicas actualizando AC y CC.
13. Ejecuta operaciones de memoria, pila, control de flujo y DMA según el opcode.
14. Solicita interrupciones cuando ocurre un evento excepcional.
15. Registra eventos relevantes en el log.
16. Retorna éxito o error según el resultado del ciclo.

Utiliza: handle_interrupt, mmu_translate, bus_read, get_value, cpu_interrupt, write_log

Referenciada en: main

DISK

Este módulo representa la unidad física de un disco magnético de almacenamiento masivo. Se implementó como un arreglo de 4 dimensiones, las tres primeras representan la pista, cilindro y sector respectivamente, cada sector almacena una palabra del sistema. La última dimensión se corresponde con un arreglo de caracteres, donde cada posición representa un carácter de la palabra almacenada en el sector, 8 dígitos + '\0'. Su funcionamiento radica en leer y escribir sectores del disco.

int disk_init(void)

Parámetros: void

Descripción: Inicializa el disco. Devuelve -1 en caso de error, 0 de lo contrario

Implementación:

1. Inicializar todas las posiciones del arreglo del disco en 0
2. Inicializar el mutex que protege el acceso a las secciones críticas del código del disco. Retorna -1 en caso de no poder crearlo, de lo contrario continua

Referenciada en: **system_init**

Utiliza: **write_log**

void disk_destroy(void)

Parámetros: void

Descripción: Elimina toda la estructura de datos que representa al disco

Implementación:

1. Destruye el mutex inicializado en disk_init()

Referenciada en: **main**

Utiliza: none

int disk_read_sector(int track, int cylinder, int sector, char *out_buf)

Parámetros:

- int track: número de la pista a leer
- int cylinder: número del cilindro a leer
- int sector: número del sector a leer
- char *out_buf: Word leída

Descripción: Lee un sector del disco

Implementación:

1. Valida los Parámetros. Retorna -1 en caso de error
2. Bloquea el acceso al disco usando el mutex
3. Copia la cadena de caracteres guardada en el sector al out_buf
4. Desbloquea el acceso al disco usando el mutex. Retorna 0

Referenciada en: **dma_perform_io**

Utiliza: **write_log**

int disk_write_sector(int track, int cylinder, int sector, const char *in_buf)

Parámetros:

- int track: número de la pista a escribir
- int cylinder: número del cilindro a escribir
- int sector: número del sector a escribir
- char *in_buf: Word a escribir

Descripción: Escribe un sector del disco

Implementación:

1. Validar Parámetros. Retorna -1 en caso de error
2. Bloquear el acceso al disco usando el mutex
3. Copiar el contenido del buffer de entrada al sector
4. Desbloquear el acceso al disco

Referenciada en: **dma_perform_io**

Utiliza: **write_log**

DMA

Este módulo trabaja como la vía de comunicación y operación de las instrucciones E/S. Su estructura de datos está definida en brain.h. Este recibe todas las instrucciones E/S directo del CPU gracias a la función dma_handler la cual después de recibir la instrucción *sdmaon*, crea un hilo de trabajo que ejecutará una operación asíncrona, al finalizar le envía una interrupción al procesador (código 4 – INT_IO_END).

Este hilo de trabajo utiliza directamente la estructura de datos del DMA, después de validar Parámetros de entrada verifica si la operación es de input o de output:

1. IO = 0 (copia el valor de la memoria y lo formatea y lleva al disco)
2. IO = 1 (copia el valor del disco y lo formatea y lleva a la memoria)

int dma_init()

Parámetros: void

Descripción: Inicia el módulo DMA

Implementación:

1. Chequear si el DMA ya fue inicializado. Simula una especie de Singleton
2. Inicializar un mutex para proteger el acceso al módulo. Retorna -1 en caso de error
3. Inicializar todos los valores de la estructura de datos del DMA
4. Retorna 0

Referenciada en: **system_init**

Utiliza: **write_log**

void dma_destroy()

Parámetros: void

Descripción: Elimina el módulo DMA

Implementación:

1. Chequea si el DMA ya fue inicializado, si no lo está simplemente retorna.
2. Chequea si el hilo que ejecuta la operación E/S de manera asíncrona se está ejecutando. De ser así, se espera a que termine su ejecución y se elimina el hilo.
3. Se elimina el mutex
4. Se indica que el DMA ya no está inicializado y no hay un hilo de trabajo en ejecución

Referenciada en: **system_init**

Utiliza: **write_log**

int dma_handler(int opcode, int value, unsigned int mode)

Parámetros:

- int opcode: código de operación
- int value: valor que va a usar la operación

- unsigned int mode: modo de ejecución de la operación (kernel o usuario)

Descripción: Recibe todas las instrucciones de E/S directamente desde el procesador, configura el DMA acorde a los valores recibidos. Crea un hilo de trabajo para realizar una operación E/S asíncrona.

Implementación:

1. Chequea si el DMA ya fue inicializado, si no lo está simplemente retorna -1
2. Bloquea el mutex del DMA antes de acceder a la estructura de datos
3. Verifica el opcode con un switch, configura la estructura de datos dependiendo de la operación
4. Si la operación es sdmaon, se verifica que el DMA no esté ocupado. En caso de estarlo se retorna DMA_BUSY_CODE (99)
5. Se validan las direcciones de memoria ram y Parámetros del disco. Retorna -1 en caso de error.
6. Se indica explícitamente que el DMA ahora está ocupado editando la estructura de datos
7. Se libera el mutex del DMA antes de crear el hilo de trabajo para evitar bloquear el mutex
8. Se crea el hilo dma_perform_io el cual arrancará su ejecución. En caso de error se bloquea el mutex nuevamente y se libera el DMA, se desbloquea el mutex y retorna -1
9. Retorna 0

Referenciada en: **cpu**

Utiliza: **write_log**, **dma_perform_io**

void *dma_perform_io(void *arg)

Parámetros:

- void *arg: un puntero genérico a los datos que va a usar el DMA, como este está definido globalmente nada más se colocó para no tener errores. En resumen, no hace nada.

Descripción: Realiza las operaciones E/S, se comunica con la memoria y el disco para realizar las transferencias.

Implementación:

1. Adquiere exclusión mutua para acceder al DMA
2. Valida los Parámetros a utilizar del disco
3. Chequea si la operación es de input (disco->memoria) o output (memoria->disco)
    a. CASO A: memoria -> disco (IO = 0).
        i. Se lee la memoria accediendo al bus del sistema. Si falla se setea el estado del dma (dma.STATE=1, dma.BUSY=0) y retorna.
        ii. Formatea el Word leído a una cadena de caracteres
        iii. Escribe esa cadena en el disco en la posición configurada. Si falla se setea el estado del dma (DMA.state=1, dma.BUSY=0) y retorna
        iv. Setea el DMA (dma.STATE=0, dma.BUSY=0) y libera el mutex
        v. Envía una interrupción al procesador (INT_IO_END)

    b. CASO B: memoria -> disco (IO = 0).
        i. Se lee el disco. Si falla se setea el estado del dma (dma.STATE=1, dma.BUSY=0) y retorna.
        ii. Formatea la cadena de caracteres recuperada a un Word de la arquitectura
        iii. Escribe ese Word en la posición configurada. Si falla se setea el estado del dma (DMA.state=1, dma.BUSY=0) y retorna
        iv. Setea el DMA (dma.STATE=0, dma.BUSY=0) y libera el mutex
        v. Envía una interrupción al procesador (INT_IO_END)
4. Retorna

Referenciada en: **dma_handler**

Utiliza: **write_log**, **bus_read**, **bus_write**, **disk_read_sector**, **disk_write_sector**

int dma_is_busy()

Parámetros: void

Descripción: Indica si el DMA está trabajando

Implementación:

1. Chequea si el DMA ya fue inicializado, si no lo está simplemente retorna 0.
2. Bloquea el mutex del DMA para acceder a dma.BUSY y almacenarlo en una variable temporal
3. Desbloquea el mutex del DMA
4. Retorna la variable temporal (busy=0, not busy=1)

Referenciada en: none

Utiliza: none

int dma_get_state()

Parámetros: void

Descripción: Indica si hubo un error durante la operación E/S (error=1, éxito=0)

Implementación:

1. Chequea si el DMA ya fue inicializado, si no lo está simplemente retorna 1.
2. Bloquea el mutex del DMA para acceder a dma.STATE y almacenarlo en una variable
3. Desbloquea el mutex
4. Retorna el estado

Referenciada en: **cpu**

Utiliza: **write_log**