Colección de Problemas de Virtualización

Garantía y Seguridad. Universidad de Zaragoza

22 de mayo de 2026

Notas preliminares

Esta breve colección de problemas corresponde a la Asignatura Garantía y Seguridad impartida en el octavo semestre del grado en Ingeniería Informática de la Universidad de Zaragoza.

Si detecta cualquier errata en alguno de los ejercicios planteados o tiene alguna sugerencia de mejora por favor comuníqueselo a Darío Suárez Gracia, dario@unizar.es.

Problemas

1. Una máquina virtual esta corriendo sobre un hipervisor de tipo 1. La ISA de la máquina virtualizada, VM, y la del hardware real sobre el que corre el hipervisor son la misma y todas las instrucciones sensibles son privilegiadas. Dicha máquina dispone de 2 modos de ejecución: privilegiado y usuario.
    1. Indica por favor si el sistema operativo de la VM se ejecutará siempre en el mismo modo e indica cuál es en caso afirmativo.
    2. Una aplicación de la VM ejecuta una llamada al sistema. Dibuja un diagrama en el que figuren las interacciones que se producen entre la aplicación y el sistema operativo de la VM con el hipervisor y el hardware real, incluyendo el control de las interacciones. En el diagrama se deberá mostrar claramente cual es el nivel de privilegio de ejecución de la aplicación y el sistema operativo de la VM y del hipervisor.
    3. ¿Cuántos cambios de contexto se producen en el diagrama anterior?
    4. Si la llamada al SO es un brk (brk/sbrk aumentan y disminuyen el tamaño del segmento de datos de un proceso), ¿quién almacenará el cambio en el segmento de datos, el hipervisor o el sistema operativo de la VM?
    5. Para posibilitar la lectura de contadores hardware, se ha añadido a la ISA una nueva instrucción HCSWI (hardware-counters software interrupt) que al ejecutarse en modo usuario recibe un parámetro que es el número de contador a leer y escribe en un registro el valor leído. En modo privilegiado recibe 3 parámetros: número de contador, lectura/escritura, y valor. Cuando el segundo parámetro es lectura, la instrucción funciona igual que en el caso anterior. Pero si el segundo parámetro es escritura, la instrucción escribe el valor del tercer parámetro en el contador. El sistema operativo utiliza los 3 parámetros para poner a cero los registros cuando un proceso empieza su ejecución. Indica si esta nueva instrucción puede suponerle algún problema al hipervisor de cara a la virtualización. En caso de que la respuesta sea afirmativa, explica por favor como podría solucionarse el problema.
2. Los traductores binarios permiten traducir instrucciones de un repertorio a otro y suelen utilizar caches de código para mejorar el rendimiento. Dado el siguiente programa en C++:

```
const size_t N = 10;
int a[N];

for(size_t i = 0; i < N; i++) {
    if (i & 0×1) {
        a[i]=i;
    } else {
        a[i] = N - i;
    }
}
```

Y su correspondiente ensamblador en i386:

```
    movl $0, %eax
    movl $10, %ecx
.L5:
    testb $1, %al
    je .L2
    movl %eax, a(,%eax,4)
    jmp .L3

.L2
    movl %ecx, %edx
    subl %eax, %edx
    movl %edx, a(,%eax,4)
.L3
    addl $1, %eax
    cmpl $10, %eax
    jne .L5
```

Indica cómo serían traducidos los bloques básicos[1] por un traductor binario. Un bloque básico es un conjunto de instrucciones con un único punto de entrada y de salida. Respecto a la arquitectura destino, debes emplear la arquitectura ARM v7 que ofrece 16 registros de propósito general, desde r0 a r15. Los registros r13, r14 y r15 están reservados para el puntero de pila, el registro de enlazado y el contador de programa. Para que la conversión entre i386 y ARM os resulte más fácil podeis usar la siguiente tabla de conversión:

i386	ARM v7	Notas
`movl`	`mov/ldr/str`	`mov`: operandos registros, `ldr/str`: direcciones de memoria
`testb`	`tst`	And bit a bit
`subl`	`sub`	restar operados
`jmp`	`b`	Salto incondicional
`jne`	`bne`	Salto condicional

Para cargar la dirección inicial de una variable global en ARM se puede emplear la instrucción `lea rX, =a`.

Asumiendo que utilizas una cache de código que opera por bloques básicos, siendo un bloque básico una secuencia de instrucciones entre saltos, indica:

[1] Un bloque básico es una secuencia ordenada de instrucciones que sólo tienen un único punto de entrada y de salida. Es decir, únicamente la última instrucción puede ser una instrucción de salto. Por ejemplo, las instrucciones siguientes forman un bloque básico: `movl $0,%eax; movl $10,%ecx`.

    1. ¿Cuáles serían los bloques básicos del código en i386 y luego tradúcelos a ARM?
    2. Una vez traducidos, escribe la secuencia de ejecución tras completarse 2 iteraciones. Puedes identificar cada bloque con una letra mayúscula y escribir la cadena resultante.
    3. ¿Podría un traductor binario más inteligente traducir completamente este código a granularidad de función en vez de bloque básico?
    4. Asumiendo que la arquitectura i386 sólo tuviera las instrucciones que aparecen en el código anterior, indica si sería una arquitectura virtualizable o no. En caso afirmativo, indica alguna otra instrucción del repertorio i386 que haga la arquitectura no virtualizable y explica la causa.
    5. ¿Se podría utilizar el mecanismo de traducción binaria una vez "atrapada" la instrucción sensible y así traducirla?

3. Explica dónde guarda un hipervisor el estado, registros de propósito general, flags, ... de la maquina virtualizada. Si la respuesta incluye soporte hardware, ¿podrías indicar si son necesarias instrucciones adicionales para ayudar al hipervisor?
4. Una empresa quiere mejorar su centro de datos y disponen de 2 tipos de sistemas a virtualizar: uno en el código fuente esta disponible y otro en el que no (el sistema es propietario). Explica brevemente si esta empresa debería optar por hipervisores tipo 1, 2 y/o paravirtualización para ambos tipos de máquinas.
5. Una empresa ofrece servicios de virtualización en la nube para la ejecución de máquinas virtuales ARM. Las máquinas virtuales se ejecutan en un hipervisor de tipo 1 corriendo a su vez sobre procesadores ARM v7a capaces de ejecutar instrucciones vectoriales NEON. Lamentablemente, el proceso de fabricación de estos procesadores ha tenido fallos intermitentes y en algunos de los procesadores al ejecutar cualquier instrucción NEON como VADD, vector add, salta una excepción de instrucción inválida.
    1. La empresa desea poder ejecutar y migrar las máquinas virtuales entre todos los nodos independientemente si NEON esta activo o no. Explica por favor de manera clara y concisa cómo podría permitirse la ejecución de las máquinas virtuales en cualquier procesador, funcione o no funcione la unidad NEON, sin modificar el binario.
    2. Uno de los códigos que corren en las máquinas virtuales suma dos vectores en un tercer vector tal y como muestra el siguiente código en C++:

```
#include <arm_neon.h>
#include <cstddef>

void suma_vectores(const uint32_t restrict *a,
        const uint32_t restrict *b,
        uint32_t restrict *c,
        const size_t n)
{
    // n es siempre múltiplo de 4
    for(size_t i = 0; i < n; ++i) {
        c[i]=a[i]+b[i];
    }
}
```

    Y su correspondiente traducción en ensamblador es cuando n es múltiplo de 4 (se puede asumir que el tercer argumento de suma_vectores ha sido dividido por 4):

```
suma_vectores(uint32_t const*, uint32_t const*,
        uint32_t **, unsigned int):
```

```
        cmp r3, #0
        bxeq    lr
        mov     ip, #0
.L4:
        vld1.32 {d18-d19}, [r0]!
        vld1.32 {d16-d17}, [r1]!
        vadd.i32        q8, q8, q9
        add     ip, ip, #1
        cmp     ip, r3
        vst1.32 {d16-d17}, [r2]!
        bne     .L4
        bx      lr
```

Asumiendo que el hipervisor emplea interpretación básica, indica que instrucciones serían interpretadas y su correspondiente código en ensamblador que sería ejecutado en una máquina sin NEON. Para las instrucciones repetidas sólo hay que hacerlo una vez.

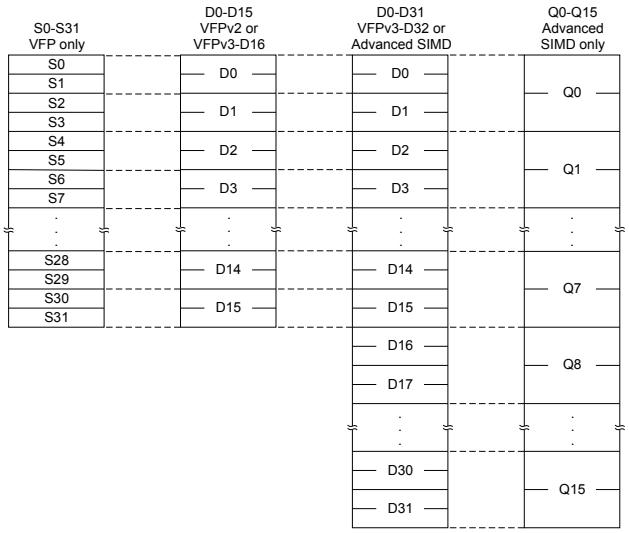

Figura 1: Banco de registros ARM NEON

El banco de registros de ARM NEON esta organizado en 32 entradas de 64 bits que pueden ser accedidas como 16 entradas de 128 bits (Q0-Q15) o 32 entradas de 64 bits (D0-D31). Por ejemplo, la instrucción `vld1.32 {d16-d17}, [r0]!` lee 128 bits de memoria y los guarda en los registros d16 y d17 o q8.

Nota, el hipervisor garantiza que cuando empieza la interpretación de cualquier instrucción hay un registro en memoria denominado `regs` que almacena el valor de todos los registros del sistema. Además puedes asumir que la rutina del intérprete guarda todos los registros del procesador en la pila y que los desapila al terminar. De cara a utilizar los registros, se dispone de etiquetas denominadas `reg_mem` que apuntan a la dirección de memoria de cada uno de los registros de la máquina en el struct `regs`. Por ejemplo `ldr r0, =r0_addr` carga la dirección de memoria de `regs.r0` en el registro `r0` del procesador. Para simplificar la decodificación, puedes utilizar las funciones `get_reg_src()` y `get_reg_dst()` que devuelven en r0 (y r1, r2, ... cuando sea necesario) los registros que emplee la instrucción que actualmente se está interpretando. El valor devuelto es el offset en bytes relativo al struct `regs`. En el código puedes escribir `bl get_reg_src()` por ejemplo. Cuando la instrucción emplee varios registros, la función tendrá múltiples registros destino.

```
struct regs {
    uint32_t r0;
    uint32_t r1;
    ...
    uint32_t r15;
    ...
}
```

3. ¿Debería pertenecer el banco de registros NEON al estado del procesador de la máquina virtualizada? Y si se migra la máquina a otro nodo, ¿se copiará el banco?

6. Se han añadido nuevas instrucciones a un procesador, cuya ISA es virtualizable, para acceder en el registro contador de ciclos del procesador, CYCLEREG. Este registro se pone a cero al arrancar la máquina, se incrementa en cada ciclo de ejecución y lo puede usar el sistema operativo para distribuir de manera equitativa los ciclos de procesador.

   1. La instrucción `ldc rsX` permite leer el valor del registro CYCLEREG y almacenarlo en la posición de memoria apuntada por el registro `rsX` tanto en modo usuario como de sistema. Indica por favor si una vez añadida esta instrucción la ISA sigue siendo virtualizable. Justifica brevemente la respuesta.

   2. La instrucción `stc rsX` permite escribir el valor del registro CYCLEREG desde la posición de memoria apuntada por `rsX` tanto en modo usuario como de sistema. Indica por favor si una vez añadida esta instrucción la ISA sigue siendo virtualizable. Justifica brevemente la respuesta.

7. La arquitectura VZ-JIT presenta la instrucción SVC para realizar llamadas al sistema. El formato de la instrucción es `SVC #imm` donde `#imm` puede tomar valores desde 0 a 63. Al ejecutarse salta a la entrada correspondiente del vector de excepciones.

   1. Un hipervisor con paravirtualización quiere utilizar las entradas desde la 64 a a la 127 (originalmente libres) para ejecutar las llamadas al hipervisor. Es decir, si en la entrada 0 esta la llamada al sistema `read` en el sistema operativo original, en la entrada 64 estará la llamada a `read_hipervisor` y así para todas las llamadas al sistema.

      Indica brevemente si al cargar en memoria un binario de una máquina invitada sería posible re-escribir el campo `#imm` sumándole 64 para utilizar el binario sin recompilar con paravirtualización.

   2. En caso afirmativo, presentaría esta solución alguna mejora respecto a la implementación Trap&Emulate vista en clase.

   3. En caso de disponer del código fuente del sistema operativo de la máquina invitada, sería mejor traducir el binario del sistema operativo o modificar el código fuente de las aplicaciones.

8. Se dispone de un intérprete en trama directa con predecoder que va a ejecutar el siguiente código en c con su correspondiente ensamblador ARM v7:

```
int find_max(const int *v, int n)
{
   int max = -1;

    for(int i = 0; i < n; ++i) {
        if (v[i] > max) {
            max = v[i];
        }
    }
```

```
    return max;
}

find_max:
    cmp r1, #0
    ble .L4
    mov r3, r0
    add r1, r0, r1, asl #2
    mvn r0, #0
.L3:
    ldr r2, [r3]
    cmp r0, r2
    movlt r0, r2
    add r3, r3, #4
    cmp r3, r1
    bne .L3
    bx  lr
.L4:
    mvn r0, #0
    bx  lr
```

1. Indica por favor si las instrucciones se rellenarían en la tabla de código intermedio. En caso afirmativo, y asumiendo el siguiente formato:

   ```
   typedef struct {
       uint32_t op;
       uint8_t dest, src1, src2;
       uint8_t cond_codes;
   } instruction_t;

   instruction_t code[MAX_BINARY_SIZE];
   ```

   Indica el rellenado de 3 instrucciones después de la etiqueta .L3. Pueden definirse tipos enumerados, `enum` para representar los códigos de operación, los registros y los códigos de condición. En otras palabras, debe hacerse manualmente el trabajo del predecoder. En caso de que un campo deba quedarse vacío, se puede emplear el valor NULL.

2. Escribe por favor la rutina del intérprete para la instrucción `add` en lenguaje c.
3. Si en la rutina del interprete has empleado más de un contador de programa, explica por favor la razón.
4. Podría utilizarse el código del intérprete para ejecutar el binario ARM en una arquitectura PowerPC.

9. En la arquitectura VT-JIT, se añade una nueva instrucción sensible `FCH #set`, flush cache, que invalida todas las vías del conjunto `#set` cuando se ejecuta en modo usuario y elimina todas las vías de todos los conjuntos en modo privilegiado tal y como muestra la figura.
   1. Podría plantear algún problema esta nueva instrucción de cara a la virtualización.
   2. Si plantea algún problema, indica por favor algún mecanismo hardware que permitiera resolverlo.
   3. Podría emularse el comportamiento de la instrucción en modo privilegiado en modo usuario. En caso afirmativo, explica cómo.

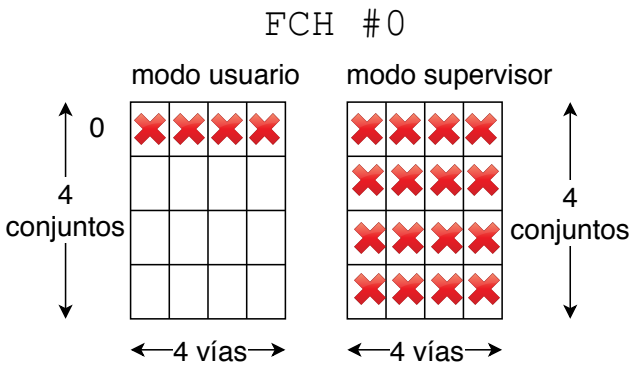

Figura 2: Ejemplo de FCH #set

4. La arquitectura VT-JIT dispone de 2 modos de funcionamiento, podría un nuevo tercer módo, hipervisor, resolver el problema.

10. Dado el siguiente diagrama, blktap, indica cuantas llamadas al sistema se realizarán al menos cuando desde el DomU realice una lectura de un único bloque de un fichero y quién la está realizando (el hipervisor o la máquina virtual).

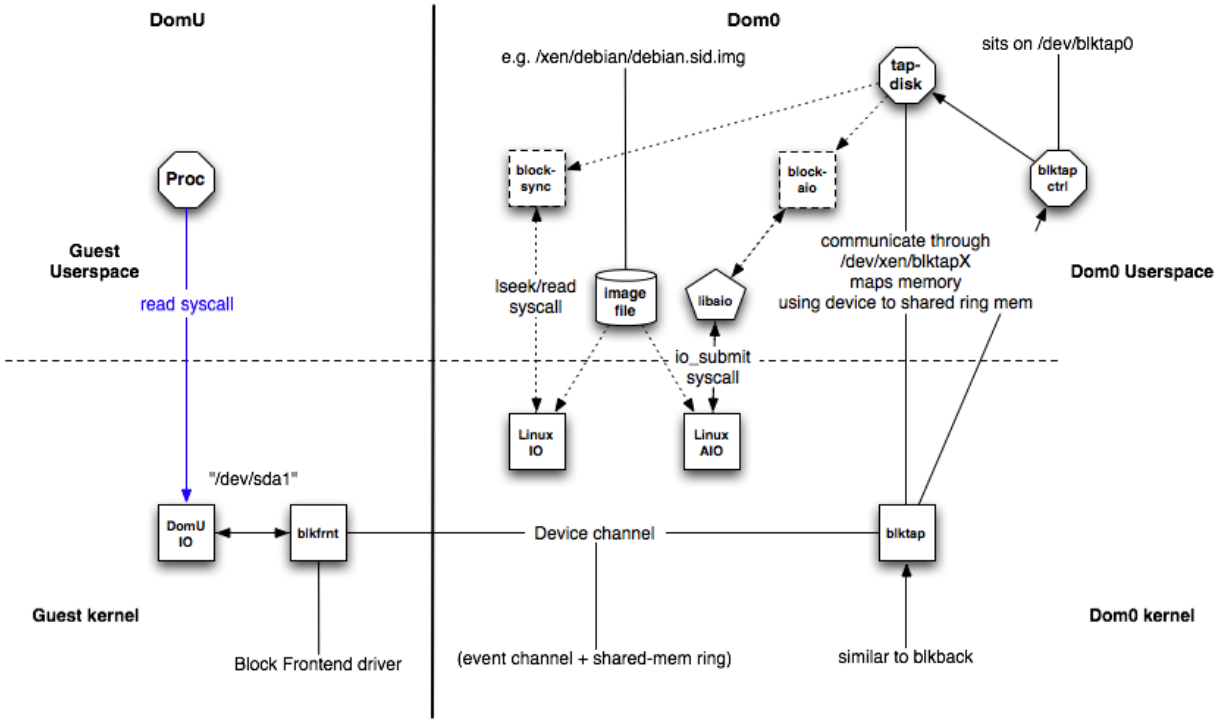

Figura 3: Diagrama de actividades durante un blktap

11. Un fabricante de procesadores desea implementar un procesador capaz de realizar traducción binaria hardware entre ARM v7 y RISC-V. Para probar el traductor necesita conocer la traducción del siguiente código C/ARM v7 a RISC-V.

A continuación se muestra el código en C y el código ensamblador ARM v7:

```
int find_min(const int *array, const size_t n) {
  int min = INT_MAX;

  for(size_t i = 0; i < n; ++i) {
```

```
    if (array[i]< min) {
      min = array[i];
    }
  }
  return min;
}
```

```
cmp     r1, #0
mov     r3, r0
mvn     r0, #-2147483648
bxeq    lr # instrucción predicada sólo se ejecutará si r1 es igual a cero
sub     r3, r3, #4
add     r1, r3, r1, lsl #2
.L3:
ldr     r2, [r3, #4]!
cmp     r0, r2
movge   r0, r2
cmp     r3, r1
bne     .L3
bx      lr
```

1. ¿Indica por favor cuales son los bloques básicos del código ARM v7?
2. Cuándo la variable n tome el valor 2, ¿cuál será la secuencia de ejecución de los bloques básicos? Cada bloque debe identificarse con una letra mayúscula empezando por la A.
3. Asumiendo la siguiente conversión de instrucciones entre ARM v7 y RISC-V, realiza la traducción de las intrucciones entre la etiqueta `.L3` y el final de la secuencia, instrucción `bx lr`. Los registros en RISC-V van desde x0 (siempre tiene el valor 0) a x31.

ARM v7	RISC-V	Notas
ldr	lw	Cargar de memoria
cmp	ble	Los saltos condicionales en RISC-V realizan operaciones aritméticas. Por ejemplo: ble <fuente0><fuente1>,<destino>
mov	mv	Mover operandos entre registros
bne	bne	Salto si no es igual
bx	ret	Retorno de subrutina

4. ¿Podría virtualizarse el código ARM v7 anterior empleando trap&emulate? En caso afirmativo, podrías añadir alguna instrucción al repertorio que hiciera el código no virtualizable?

12. Un fabricante de procesadores desea implementar un procesador capaz de realizar traducción binaria hardware entre ARM v7 y RISC-V. Para probar el traductor necesita conocer la traducción del siguiente código C/ARM v7 a RISC-V RV32I.

    A continuación se muestra el código en C y el código ensamblador ARM v7:

```
void insertion_sort(int32_t *array, int32_t n) {
    for(int32_t i = 1; i < n; ++i) {
        int32_t elem = array[i];
        int32_t j;
```

```
        for(j = i-1; (j >= 0) && (array[j] > elem); --j) {
          array[j+1] = array[j];
        }
        array[j+1] = elem;
    }
}

insertion_sort(int*, int):
        cmp     r1, #1
        ble     .L10
        push    {r4, r5, lr}
        mov     r5, r0
        mov     lr, r0
        subs    r4, r1, #1
        mov     ip, #0
        b       .L6
.L4:
        adds    r3, r3, #1
        str     r0, [r5, r3, lsl #2]
        add     ip, ip, #1
        cmp     ip, r4
        beq     .L13
.L6:
        ldr     r0, [lr, #4]!
        mov     r3, ip
        mov     r2, lr
.L3:
        ldr     r1, [r2, #-4]
        cmp     r1, r0
        ble     .L4
        str     r1, [r2], #-4
        subs    r3, r3, #1
        bpl     .L3
        b       .L4
.L13:
        pop     {r4, r5, pc}
.L10:
        bx      lr
```

1. ¿Indica por favor cuales son los bloques básicos del código ARM v7?
2. Cuando al vector `array` y a la variable `n` se les asignen los siguientes valores: `array[2]={2,1}` y `n=2`, ¿cuál será la secuencia de ejecución de los bloques básicos? Cada bloque debe identificarse con una letra mayúscula empezando por la `A`.
3. Asumiendo la siguiente conversión de instrucciones entre ARM v7 y RISC-V, realiza la traducción de las intrucciones entre la etiqueta .L3 y el final de la secuencia, instrucción `b .L3`. Los registros en RISC-V van desde x0 (siempre tiene el valor 0) a x31.

ARM v7	RISC-V	Notas
ldr	lw	Cargar de memoria
str	sw	Escribir en memoria
add	add	Suma de operandos
sub	sub	Resta de operandos

ARM v7	RISC-V	Notas
mov	mv	Mover operandos entre registros
cmp	ble	Los saltos condicionales en RISC-V realizan operaciones aritméticas. Por ejemplo: ble <fuente0><fuente1>,<destino>
mov	mv	Mover operandos entre registros
bne	bne	Salto si no es igual
bge	bge	Salto si mayor o igual
bx	ret	Retorno de subrutina

Notas:

- El registro `ip` en ARM es `r12`. En este código `ip` se emplea como registro temporal no-preservable. Cuando una función no invoca a otra función, como en este caso, el registro `lr` también puede ser empleado como registro no-preservable, al igual que `r[0-3]`. Los accesos a memoria post-incremento como `ldr ip, [r4], \#4`, cargan el contenido apuntado por el registro origen, `r4`, en el registro destino, `ip`, y después le suman el inmmediato al registro origen, `r4 = r4 + 4`.
- Los registros en RISC-V van desde `x0` (siempre tiene el valor 0) a `x31`. Se asumirá que RISC-V RV32I no dispone de modos de direccionamiento con pre y post-incremento.

4. Un sistema operativo emplea `insertion_sort()` para ordenar la lista de procesos listos para ejecución y lanzar los más prioritarios. Para acelerar el propio kernel, se va a añadir una instrucción al repertorio RISC-V: `INS rs1, rs2`, que ordena el vector de longitud `rs2` almacenado en la dirección `rs1`. Asumiendo que la arquitectura RISC-V es virtualizable, indica si la nueva instrución `INS` hace que deje de serlo.
5. Además de la instrucción `INS`, se plantea añadir la instrución `INSSW`: insertion sort and switch, que además de ordenar la lista de procesos, en modo supervisor, guarda el índice del proceso más prioritario y directamente hace el cambio de contexto. ¿Sería virtualizable la arquitectura resultante en este caso? En caso afirmativo, podrías añadir alguna instrucción al repertorio que hiciera el código no virtualizable?

13. Un fabricante de hardware desea implementar para sus microcontroladores un interprete en trama directa de la arquitectura RISC-V RV32I y desea comprobar si su predecoder es correcto. Podrías por favor ayudándoles escribiendo el código predecodificado de la siguiente secuencia de instrucciones que incrementa el tope de la pila según el contenido del registro t1.

```
lw t0, (sp)
add t0, t0, t1
st t0, (sp)
```

Se puede asumir que el código intermedio tiene el siguiente formato:

```
typedef struct {
unsigned int op_code;
unsigned int dst, src0, src1;
} instruction;
```

1. ¿Es necesario que el intérprete guarde el estado del procesador emulado en memoria? En caso afirmativo, para el banco de registros, asume que t0, t1 y sp se almacenan en las posiciones 2, 5 y 6 de los registros físicos de la arquitectura.

2. ¿Podrías rellenar el vector del código intermedio asignando los valores que consideres al `op_code`?
3. Si se añade una instrucción sensible no-privilegiada, ¿supondría algún problema para el intérprete?

14. La Arquitectura ARMv8 define varios tiempos para sistemas con paravirtualización que se corresponden aproximadamente con los tiempos del interfaz VMI: real, robado y disponible (Apuntes Tema 4, transparencia 31). En un hipervisor ejecutándose en un mono-procesador donde se han lanzado 2 máquinas virtuales paravirtualizadas, asumiendo la siguiente lista de eventos de scheduling, indica por favor cuáles serían los tiempos reales, robados y disponibles de ambas. Se deberá asumir que el tiempo real de una VM empieza con su creación y termina con su apagado.

Tiempo (s)	MV	Evento
2	VM1	creación VM
2	VM1	estado VM running
6	VM1	estado VM halted
6	VM2	creación VM
6	VM2	estado VM running
10	VM1	estado VM ready
12	VM2	estado VM halted
12	VM2	terminación VM
14	VM1	estado VM running
16	VM1	estado VM halted
16	VM1	terminación VM

Para la resolución se pueden realizar 2 líneas temporales con los estados de las máquinas virtuales a partir de la siguiente figura y con ambas calcular los tiempos.

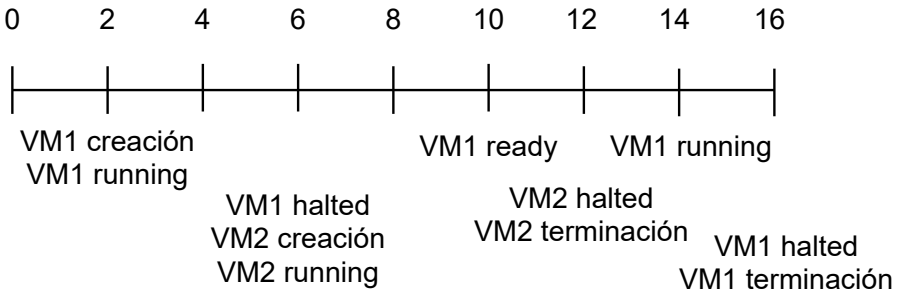

Figura 4: Diagrama temporal de los eventos

15. Un fabricante de procesadores RISC-V ha realizado dos modificaciones en sus chips y desconoce si los cambios mantienen la arquitectura virtualizable.

    1. Para aumentar la fiabilidad de sus procesadores sin comprometer el consumo energético sus unidades aritmético-lógicas, ALU, se comportan de manera diferente según el modo del procesador. Cuando el procesador se encuentra en modo U, todas las operaciones se realizan de manera estándar. Sin embargo, si el procesador está en modo M o S, internamente, sin modificar la ISA y sin hacer visible el mecanismo al programador, las operaciones en la ALU se realizan 3 veces y se almacena el resultado mas probable. Por ejemplo, al ejecutar la instrucción `add x3, x1, x2` asumiendo que los registros `x1` y `x2` almacenan los valores 4 y 5, la suma se calculará 3 veces y si el resultado de dichas sumas es de 9, 8, 9, en el registro `x3` se escribirá el valor 9.

¿Podrías por favor indicar brevemente si este cambio ha hecho que la arquitectura deje de ser virtualizable?

2. Realizando mediciones han visto que las llamadas al sistema implementadas con la instrucción `ECALL` pueden ser muy costosas, especialmente para leer timers. Para acelerarlas han añadido una nueva instrucción `FASTT rd, rs1, #imm` con la siguiente semántica: si `rd` es igual a `x0` y el procesador está en modo S, entonces la instrucción escribe el valor almacenado en `rs1` mas `#imm` en la dirección donde esté mapeado el timer de sistema. Y si el procesador está en modo U, la instrucción guarda el valor del timer del sistema en el registro `rd` siempre que sea distinto de `x0`. ¿Podrías indicar: a) si la nueva instrucción mantiene la ISA virtualizable en sistemas RISC-V sin y con la extensión H? Y, b) ¿en qué tipo de instrucción se podría codificar `FASST`?

16. Para el segmento *High Performance Computing*, el mismo fabricante ha añadido una nueva instrucción reduce add, `RADD`. Esta instrucción suma hasta 4 registros consecutivos acumulando el resultado en `rd`. Su sintaxis es `RADD rd, rs_begin, rs_end`, siendo el `rs_begin` y `rs_end` el primer y último registro a acumular. Por ejemplo, `radd x1, x2, x5` realizaría la siguiente operación: `x1 = x2 + x3 + x4 + x5`.

    La decisión de agregar la instrucción se ha tomado viendo que la mayoría de los centros de HPC emplean muy frecuentemente este fragmento de código:

```
const size_t n = 8192;

int reduce_add_array(const int* array, size_t n) {
  int res = 0;
  for(size_t i = 0; i < n; ++i) {
    res+=array[i];
  }
  return res;
}
```

    Siendo su traducción a código ensamblador en RISC-V (rv32gc):

```
  li a2, 0
  beqz a1, .LBB0_2
.LBB0_1:
  lw a3, 0(a0)
  add a2, a2, a3
  addi a1, a1, -1
  addi a0, a0, 4
  bnez a1, .LBB0_1
.LBB0_2:
  mv a0, a2
  ret
```

    1. Indica por favor los bloques básicos del código ensamblador empleando letras mayúsculas, empezando por la A, para identificar los bloques.
    2. ¿Puede suponer la instrucción `RADD` algún riesgo para la virtualización de la arquitectura?
    3. Podría un interprete que opera instrucción a instrucción, como el implementado en prácticas, beneficiarse de la nueva instrucción `RADD`. En caso afirmativo, ¿podrías por favor indicar como sería el código de la rutina de interpretación en pseudo-c/c++?
    4. Sería capaz un traductor binario dinámico beneficiarse de la nueva instrucción `RADD`. En caso afirmativo: a) Podrías indicar si debería trabajar a granularidad de instrucción

o de bloques básicos y, b) Una vez fijada la granularidad, podrías escribir el código ensamblador resultante de la instrucción o bloque básico donde se empleara `RADD`.

Notas: Puedes asumir que `n` *será siempre múltiplo de 4 y se pueden emplear los 32 registros de la arquitectura sin respetar el ABI. Al comenzar la función, el registro* `a0` *contiene la dirección inicial de* `array` *y* `a1` *el argumento* `n`*. El tamaño del tipo* `int` *es de 32 bits. En RISC-V, a los registros* `x10-x11` *se les nombra en la ABI como* `a0-a1` *y pueden emplearse como argumentos de función y para devolver valores. A los registros* `x12-x17` *se les denomina* `a2-a7` *e igualmente se emplean para pasar argumentos.*

17. (5 puntos) Una de las últimas extensiones ratificadas del repertorio de instrucciones RISC-V es la de manipulación de bits. Esta extensión introduce la instrucción `cpop rd, rs` que cuenta el número de bits a 1 en el registro fuente, `rs`, y almacena el resultado en el registro `rd`.

    Lamentablemente, hay muchos binarios existentes que no emplean dicha instrucción y que utilizan el siguiente código en C++ basado en un bucle para contar los bits a 1.

```
uint32_t pop_count(uint32_t n)
{
    uint32_t count = 0;

    while(n≠0) {
        count+= (n & 0×1) == 0×1 ? 1 : 0;
        n = n >> 1;
    }
    return count;
}
```

    Este código se suele traducir a la siguiente secuencia de instrucciones RISC-V (rv32gc):

```
pop_count(unsigned int):
      mv a5,a0
      li a0,0 // load inmediate pseduo-instruction
      beq a5,zero,.L4
    .L3:
      andi a4,a5,1
      srli a5,a5,1 // logical right shift
      add a0,a0,a4
      bne a5,zero,.L3
      ret
    .L4:
      ret
```

    1. (0,5 puntos) Indica por favor si un interprete en trama indirecta que trabaje a granularidad de instrucción podría beneficiarse de la instrucción `cpop` reemplazando toda la secuencia de instrucciones anterior por dicha instrucción.
    2. (1 punto) Sabiendo que `pop_count` es un nombre de función único, podrías proponer alguna manera para que el intérprete se beneficiara de la instrucción `cpop`.
    3. (1 punto) Para aumentar el rendimiento de los enclaves seguros, se está planteando que en modo M, la instrucción `cpop` cambie su semántica e implícitamente cuente cuantos bits están a 1 en 2 registros fuente consecutivos. Por ejemplo, `cpop a4, a5` contará el número de bits a 1 en los registros `a5` y `a6` y sumará su resultado en `a4`. ¿Podrías indicar brevemente si este cambio hace que la arquitectura deje de ser virtualizable?

4. (1 punto) También se dispone de un traductor binario dinámico, pero se desconocen los bloques básicos de la función `pop_count`. Podrías por favor indicarlos empleando letras mayúsculas, empezando por la A, para identificar los bloques.
5. (1,5 puntos) Sería capaz un traductor binario dinámico beneficiarse de la instrucción `cpop`. En caso afirmativo: a) Podrías indicar si debería trabajar a granularidad de función, instrucción o de bloques básicos y, b) una vez fijada la granularidad, podrías escribir el código ensamblador resultante de la función, instrucción o bloque básico donde se empleara `cpop`. Asumiendo que el código se ejecuta sobre un procesador cuyo IPC es siempre 1, podrías indicar cual sería la ganancia posible en rendimiento si el traductor emplea la instrucción `cpop` en la invocación a la función `pop_count(4)`.

18. La extensión *Confidential VM Extension* (CoVE) de RISC-V permite la ejecución segura de máquinas virtuales de una manera diferente a las extensiones de AMD vistas en clase. Para aislar y proteger la memoria, RISC-V emplea una unidad hardware llamada *Physical Memory Protection* (PMP).

    Sin virtualización, la PMP funciona de la siguiente manera: al arrancar el sistema, el procesador comienza en modo de privilegio *machine* (M) y ejecuta el *firmware* de bajo nivel. En el modo M, el *firmware* puede añadir al PMP entradas que se corresponden con regiones de memoria al indicando: su dirección inicial, tamaño en bytes, modos de privilegio permitidos ... El aislamiento se garantiza porque una vez escritas las regiones en el PMP, el contenido del PMP no puede ser modificado hasta el siguiente reinicio del procesador. Posteriormente, el *firmware* comenzará la la carga del sistema operativo y el procesador entrará en modo *supervisor* (S). Finalmente, las aplicaciones de usuario se ejecutaran en modo *user* (U).

    1. Indica por favor si es posible crear una región que únicamente sea accesible por las aplicaciones y no por el OS. Asumiendo que la tabla tiene 3 campos (dirección inicial, tamaño y modos de privilegio), escribe una región de 4 MB que empiece en la dirección 0xCAFEBABE y sólo sea accesible en modo U.
    2. Asumiendo que la ISA RISC-V es virtualizable y que el procesador disponible no soporta la extensiones de virtualización de RISC-V, explica si se podría emplear el PMP para garantizar el aislamiento entre los accesos a memoria de: a) las máquinas virtuales entre sí y, b) las máquinas virtuales y el sistema operativo.
    3. Si hubiera algún caso en el que el aislamiento no estuviera garantizado, ¿se podría afirmar que *AMD Secure Encrypted Virtualization* sí que lo garantizaría?

19. Un proveedor de cloud ofrece máquinas virtuales con procesadores RISC-V pero todavía no dispone de hardware RISC-V y esta emulando el código RISC-V en procesadores ARMv7. Haciendo pruebas han detectado que el rendimiento y consumo energético de esta función `memcpy` vectorizada en RISC-V es pobre:

```c
#include <riscv_vector.h>

void *memcpy_vec(void *dst, void *src, size_t n) {
  // copy data byte by byte
  for (size_t vl; n > 0; n -= vl, src += vl, dst += vl) {
    vl = __riscv_vsetvl_e8m8(n);
    vuint8m8_t vec_src = __riscv_vle8_v_u8m8(src, vl);
    __riscv_vse8_v_u8m8(dst, vec_src, vl);
  }
}
```

Esta es la traducción en ensamblador para la arquitectura RISC-V rv64iv:

```
memcpy_vec:
    beq     a2,zero,.L10
.L3:
    vsetvli a5,a2,e8,m8,ta,ma
    vle8.v  v8,0(a1)
    sub     a2,a2,a5
    add     a1,a1,a5
    vse8.v  v8,0(a0)
    add     a0,a0,a5
    bne     a2,zero,.L3
.L10:
    ret
```

A continuación se muestra una tabla con la descripción mínima de las instrucciones vectoriales empleadas:

instrucción	descripción
`vsetvli`	Especifica el tipo de datos y tamaño del vector así como la máscara. En nuestro caso, se va a operar con elementos de 1 byte con una longitud de `a2` bytes, y se empleará una máscara para delimitar el número de elementos en la última iteración (ajuste del epílogo automático). Además, en el registro destino (`a5` en el ejemplo) guardará el número de elementos procesados por iteración
`vle8.v`	Carga en el registro vectorial (`v8` en el ejemplo), el contenido apuntado por el operando fuente (`0(a1)` en el ejemplo)
`vse8.v`	Escribe en la posición de memoria apuntada por el operando (`0(a0)` en el ejemplo el contenido del registro fuente (`v8` en el ejemplo)

1. Indica por favor los bloques básicos del código ensamblador empleando letras mayúsculas, empezando por la A, para identificar los bloques.
2. Escribe la secuencia de ejecución de los bloques básicos asumiendo vectores físicos de 512 bytes y un tamaño `n` de 1024 bytes.
3. Sería capaz un traductor binario dinámico de detectar la ejecución de esta función y ejecutar código ARMv7 optimizado en vez del código interpretado. En caso afirmativo, podrías escribir como debería ser ese código optimizado. Puedes emplear las instrucciones load y store multiple (`ldm` y `stm` de ARMv7) y asumir que el número de bytes a copiar es siempre múltiplo de 128 bytes y que el procesador ARMv7 empleado tiene una cache de instrucciones de 1 MByte de tamaño.
4. Si la instrucción `vsetvli` solo fuera posible ejecutarla en los modos M y S de la arquitectura, determina por favor si la arquitectura RISC-V dejaría de ser virtualizable.

Notas:

- *Los registros en RISC-V van desde* `x0` *(siempre tiene el valor 0) a* `x31`*. Ademas, a los registros* `x10-x11` *se les nombra en la ABI como* `a0-a1` *y pueden emplearse como argumentos de función y para devolver valores. A los registros* `x12-x17` *se les denomina* `a2-a7` *e igualmente se emplean para pasar argumentos.*