Segundo Parcial de Arquitectura de Computadoras

26 de noviembre de 2024

Instrucciones:

- Indique su nombre, apellido y número de cédula en todas las hojas que entregue.
- Las hojas deben estar numeradas y en la primer hoja debe escribirse el total de hojas entregadas.
- Apague su celular. No puede utilizar material ni calculadora.
- La duración de la prueba es de tres horas, incluyendo el tiempo para completar sus datos.

Pregunta 1 [6 puntos]

a) Describa dos de los siguientes métodos de identificación del controlador que solicita una interrupción: i) múltiples líneas IRQ independientes, ii) mecanismo INT/INTA, iii) controlador de interrupciones.

b) Si la CPU tiene una única entrada de IRQ, hay más de un dispositivo interrumpiendo y no se dispone de ninguno de los mecanismos de la parte a), explique cómo puede realizarse la identificación por software del controlador que está solicitando la interrupción.

Pregunta 2 [4 puntos]

Considere un procesador con pipeline.

a) Explique en qué consiste la técnica de predicción de saltos e indique qué tipo de hazard mitiga.
b) Describa **una** de las siguientes técnicas de predicción de saltos: i) predicción basada en la condición del salto, ii) conmutar taken/not taken, iii) Branch History Table

Pregunta 3 [5 puntos]

Sea un sistema con una memoria caché asociativa de dos vías con write back. Describa el proceso completo y **todos** los posibles casos, desde que se inicia un acceso de lectura de la memoria hasta que el dato queda disponible para la CPU y se completan todas las actividades asociadas.

Pregunta 4 [5 puntos]

a) Presente un ejemplo en MIPS de hazard WAR. Justifique.
b) Explique la técnica de register renaming y por qué soluciona este tipo de hazards.

Ejercicio 1 [20 puntos]

Una lista doblemente encadenada es una estructura de datos lineal en la que cada nodo tiene un enlace al nodo siguiente y al nodo anterior.

```
typedef struct Node {
   short data;          // Valor almacenado en el nodo
   struct Node* prev; // Puntero al nodo anterior
   struct Node* next; // Puntero al siguiente nodo
} Node;
```

a) Implementar en lenguaje C una función recursiva que recorra una lista doblemente encadenada mientras el nodo actual verifique una determinada condición. El encabezado de la función es *Node* findNode(Node* actual, char direction)*, *actual* indica el nodo que se debe procesar a continuación y *direction* determina la dirección de la recorrida (0 adelante y 1 atrás). Si la dirección es adelante se recorren los punteros next y si es atrás los punteros prev. Al retornar la ejecución el puntero actual señala al nodo que no verificó la condición. En caso que todos los nodos verifiquen la función debe retornar NULL. Para verificar la condición se utiliza la función auxiliar *verify()* que recibe un dato y devuelve si el dato recibido verifica la condición. El encabezado de la función es *char verify (short data)*.

b) Compilar la función ***findNode()*** a asembler 8086. Tanto *findNode()* como *verify()* reciben y retornan los parámetros por el stack. Todos los punteros son desplazamientos respecto de ES.

c) Determinar la cantidad máxima de nodos que puede tener la lista. Justifique su respuesta.

d) Determinar el consumo máximo de stack para una lista de N nodos. Asuma que la función *verify()* consume K palabras. Justifique su respuesta.

e) Calcular el tamaño máximo de nodos para que la función pueda ejecutarse correctamente para cualquier caso. Justifique su respuesta.

Ejercicio 2 [20 puntos]

Se desea desarrollar un controlador para una bicicleta eléctrica que permita al usuario: **i)** ajustar la potencia suministrada al motor de forma dinámica, **ii)** mostrar las revoluciones por minuto y **iii)** controlar el sistema de luces de frenado. La bicicleta incluye un botón de encendido, y el usuario podrá modificar la potencia del motor según sea necesario mientras está encendida. Se dispone de un sensor de efecto hall que detecta el campo magnético producido por un imán colocado en uno de los rayos de la rueda delantera, a los efectos de determinar las revoluciones de las ruedas. La bicicleta dispone de un display donde se debe mostrar las revoluciones por minuto (RPM) realizadas **en los últimos** 10 segundos (es decir que, en todo momento, el cálculo de las RPM se hace considerando lo sucedido en los 10 segundos previos). El sistema de luces de freno debe hacer titilar las luces con una cadencia de 500 ms mientras se presionan los frenos. Las luces deben mantenerse titilando durante 5 segundos luego de liberados los frenos. Tanto la presentación de las RPM como el sistema de luces sólo actúan cuando la bicicleta está encendida.

El puerto de entrada/salida solo lectura **STATUS** de 16 bits se estructura de la siguiente manera:
- bit 0: estado del botón de encendido (0 no presionado, 1 presionado)
- bit 1: estado del freno (0 no presionado, 1 presionado)
- bit 8 a bit 2: potencia seleccionada por el usuario (entero sin signo en el rango 0 a 99)

El puerto de entrada/salida solo escritura **CONTROL** de 8 bits se estructura de la siguiente manera:
- bit 0: luz de freno (0 apagada, 1 encendida)
- bit 7 a 1: potencia (entero sin signo en el rango 0 a 99)

Se cuenta con un puerto de entrada/salida solo escritura **DISPLAY** de 8 bits en formato BCD (rango 00 a 99).

El sensor de efecto hall interrumpe al procesador cada vez que detecta el campo magnético invocando a la rutina ***hall()***. Además, se dispone de un reloj con una frecuencia de 10 Hz que es atendido por la rutina ***timer()***.

a) Sabiendo que la máquina es dedicada, implementar el sistema propuesto en lenguaje C.
b) Compilar a asembler 8086 los aspectos de inicialización del sistema (esto es: inicialización de variables y puertos, e instalación de las interrupciones).

Respuesta 1

a) Es normal que un sistema tenga múltiples controladores de E/S. Por tanto hay que tener un mecanismo que permita saber cuál controlador de E/S generó un pedido concreto de atención mediante el mecanismo de interrupción.

- Múltiples líneas IRQ independientes: En este caso el propio CPU tiene múltiples entradas INT (numeradas, por ej: INTO, INT1, INT2, etc) y la idea es conectar un controlador de E/S a cada una de ellas. La identificación es entonces por hardware y directa: el controlador que está pidiendo la interrupción es el que está conectado a la entrada INTn donde se detecta la solicitud. En este caso hay una dirección de la rutina de servicio a la interrupción por cada línea de pedido disponible.
- mecanismo INT/INTA: La idea es que el CPU dispone de una única entrada de pedido de interrupción INT, a la cuál se conectan en modalidad OR-cableado todos los pedidos de interrupción de los distintos controladores de E/S. También dispone de una salida denominada INTA (Interrupt Acknowledge) que le avisa al controlador de E/S que ha sido aceptado su solicitud de interrupción y le indica con esa señal que coloque en el bus de datos su identificación. La CPU entonces realiza una lectura del bus de datos y obtiene e identificador. Este procedimiento se extiende para el caso de múltiples controladores mediante el encadenamiento de las señales INTA (los controladores de E/S que soportan este mecanismo tienen una entrada y una salida INTA). De esta manera cuando la CPU inicia el ciclo de atención a un pedido de interrupción y levanta su señal de salida INTA, la misma se va propagando por los distintos controladores en forma encadenada. De esta forma el primer controlador de la cadena que tenga un pedido de interrupción pendiente procederá a no continuar con la propagación de la señal INTA hacia los restantes controladores y será él quien coloque en el bus de datos su identificador.

- Controlador de interrupciones: La idea es que un controlador de interrupciones posee múltiples líneas de entrada de solicitud IRQ para la conexión de los pedidos de interrupción de los diferentes controladores de E/S. Tiene, a su vez, una única salida de pedido de interrupción INT con su correspondiente INTA para implementar el mecanismo de identificación por hardware. Cuando un controlador de E/S solicita una interrupción, el controlador de interrupciones genera el pedido a la CPU a través de la señal INT. La CPU cuando acepta la interrupción activa la señal INTA y en ese momento es el controlador de interrupciones quién coloca en el bus de datos la identificación que está asociada a la entrada IRQ por la que llegó el pedido.

b) La rutina de servicio de la interrupción debe tener una parte inicial que consista en el recorrido de los distintos controladores de E/S, leyendo los registros de estado hasta encontrar aquél que tenga su bit de "pedido de interrupción" en "1". En ese caso se invocará a la sub-rutina asociada a ese controlador específico.

Respuesta 2

a) La técnica consiste en determinar, en base a un cierto algoritmo o criterio, si el salto se va a tomar o no. Esto permite seguir haciendo el fetch de las instrucciones que con cierta probabilidad serán las que el flujo lógico del programa ejecute. Si la predicción es acertada no habrá penalización ya que no será necesario vaciar ninguna etapa del pipeline. Por tanto en este caso la eficiencia del pipeline está condicionada por la eficiencia del mecanismo de predicción del salto condicional.
Esta técnica mitiga los hazards de control que son causados por instrucciones de salto u otras modificaciones del registro IP (Instruction Pointer ó PC = Program Counter).

b)

- **Predicción basada en la condición del salto**: parte de la constatación que determinados tipos de condición del salto (que están codificados en el código de operación de la instrucción de salto) tienen mayor

tendencia a realizar el salto y otros a no realizarlo.

- **Conmutar taken/not taken**: utiliza la historia reciente de los saltos, mediante una máquina de estados que “recuerda” si el último salto fue tomado o no. La predicción debe fallar dos veces seguidas para cambiarla. El problema de la técnica es que predice el comportamiento de un salto en función de lo que hizo uno anterior que no tiene por qué ser el mismo. Por eso es que es apropiado para bucles (loops), pero no en general.
- **Branch History Table**: es un perfeccionamiento del mecanismo anterior, donde se almacena en una tabla la información de un cierto conjunto de saltos recientemente evaluados. Entonces permite realizar el algoritmo anterior sobre el comportamiento del mismo salto, sin que sea interferido por otros saltos. Esto mejora la eficiencia de la predicción.

Respuesta 3

1. Se inicia el proceso de lectura de una determinada dirección de memoria.
2. A partir de la dirección de memoria se extrae la etiqueta (TAG), el conjunto (SET) y el desplazamiento (BYTE).
3. Se accede al conjunto de la cache y dentro las dos posibles líneas correspondientes a éste se compara las etiquetas de los bloques de memoria almacenados con la etiqueta de la dirección que está siendo accedida.
4. Si se encuentra la etiqueta se produce un cache hit y se extrae de la línea coincidente el dato buscado usando el desplazamiento.
5. Si no se encuentra la etiqueta se produce un “cache miss”. Se realiza entonces el acceso a memoria para traer el bloque donde se encuentra el dato buscado y se aplica la política de reemplazo para decidir en qué línea dentro del conjunto se almacena el bloque leído. Se almacena el bloque en la línea determinada por la política. Como se utiliza write back, si la línea a reemplazar fue modificada, sus contenidos deben ser copiados a memoria principal antes de ser reemplazada.
6. Por último, se utiliza el desplazamiento para obtener el dato buscado y entregarlo a la CPU.

Respuesta 4

a) add r1, r2, r3
add r2, r4, r5

En la situación anterior, se da un hazard WAR si el registro r2 de la segunda instrucción es escrito antes de que el valor de dicho registro pueda ser leído por la primer instrucción. Este hazard puede ocurrir en procesadores con emisión fuera de orden, en la situación en que la primer instrucción esté trancada en la ventana de instrucciones (por ejemplo, esperando por el registro r3, si este está siendo modificado por otra instrucción en curso).

b) Una técnica para solucionar este problema es la técnica de register renaming. Esta técnica consiste en disponer de un banco de registros adicional, llamado banco de registros físico, el cual es invisible al programador. Al ingresar en la ventana de instrucciones, cada registro destino es mapeado a un registro físico libre. De este modo se logra evitar el hazard porque la segunda instrucción, aunque finalice antes, ya no sobreescribe el valor que debe leer la primera instrucción.

Solución Ejercicio 1

Parte a)

```
 Node* findNode(Node* actual, char direction) {
    Node* nodePtr;
    if (actual != NULL) {
        char verifica = verify(actual->data);
        if (verifica) {
            if (direction != 0) {
                nodePtr = findNode(actual->prev, direction);
            } else {
                nodePtr = findNode(actual->next, direction);
            }
        } else {
            nodePtr = actual;
        }
    } else {
        nodePtr = NULL;
    }
    return nodePtr;
}
```

Parte b)
Se asume que verify retorna el byte en la parte baja de la palabra en el stack.

Ejemplo de llamada a la función findNode

```
PUSH "actual"
PUSH "direction"
CALL findNode
POP "resultado"


NULL EQU 0

.code
findNode proc
    PUSH BP         //Preservar contexto
    MOV BP, SP
    PUSH DX
    PUSH DI

    MOV DI, [BP+6]  // DI = actual
    CMP DI, NULL    // actual != NULL
    JE actual_nulo

actual_no_nulo:
    PUSH ES:[DI]    // Push actual->data
    CALL verify
    POP DX          // DX = cond
    CMP DL, 0
    JE no_verifica

verifica:
```

```
    MOV DL, [BP+4]  // DL = direction
    CMP DL, 0
    JE next

prev:
    PUSH ES:[DI+2]  // findNode(actual->prev, direction)
    PUSH DX
    CALL findNode
    POP DX              // DX = nodePtr
    JMP retorno

next:
    PUSH ES:[DI+4] // findNode(actual->next, direction)
    PUSH DX
    CALL findNode
    POP DX     // DX = nodePtr
    JMP retorno

no_verifica:
    MOV DX, DI //nodePtr = actual
    JMP retorno

actual_nulo:
    MOV DX, NULL

retorno:
    MOV [BP+6], DX
    MOV DX, [BP+2]
    MOV [BP+4], DX
    POP DI
    POP DX
    POP BP
    ADD SP, 2
    RET
findNode endp
```

Parte c)

Cada nodo ocupa 6 bytes: dos para el dato (de tipo short) y dos para los desplazamientos. Como sólo se utilizan los desplazamientos para identificar al siguiente nodo, entonces todos los nodos deben pertenecer al mismo segmento. Un segmento son 2^16 = 65536 bytes. La dirección 0 se utiliza para representar NULL. De esta forma, la cantidad máxima de nodos es el numero n que maximiza la siguiente ecuación: 6*n <= 65536 – 1. Entonces n = 10922.

Parte d)

Cada vez que se llama a findNodo se consumen 12 bytes del stack correspondientes a:
- 2 parámetros (actual y dirección)
- IP
- 3 registros para preservar contexto (DI, DX, SP)

En el llamado de la función verify, se utilizan k+4 bytes correspondientes a:
- 1 parámetro (dato)
- IP
- k bytes que utiliza internamente la función verify

Observar que verify y findNodo, no se llaman de forma anidada, por lo que para calcular el máximo uso del stack hay

que considerar solamente la llamada a verify correspondiente al procesamiento del nodo n. De esta forma la cantidad máxima seria: 12*n+k+4.

Asumiendo una estructura con n nodos, se realizara una llamada recursiva más comprobando que prev o next según el caso sean NULL, observar que esta llamada tampoco esta anidada con verify, por lo que en este caso el máximo consumo es 12*(n+1). Es decir, el máximo total de consumo del stack será max(12*n+k+4, 12*(n+1))

Parte e)

Para poder ejecutarse en cualquier caso debe cumplir, ademas de la condición calculada en la parte c, que el stack no desborde, es decir, que pueda contenerse en un solo segmento.

Es decir que

max(12*n+k+4, 12*(n+1)) < 2^16 = 65536

A simple vista se puede observar que esta condición sera más restrictiva que la calculada en la parte c.

Solución alternativa Ejercicio 1

Parte a)

```
 Node* findNode(Node* actual, char direction) {
    if(actual != NULL && verify(actual->data)
      return findNode(direccion ? actual->prev : actual->next, direccion);
    else return actual;
}
```

Parte b)

```
NULL EQU 0

findNode proc
    POP AX  ; dir_ret
    POP DX  ; direccion
    POP BX  ; actual
    CMP BX, NULL
    JE fin
    PUSH ES:[BX]
    CALL Verify
    POP CX
    CMP CX, 0
    JNE Fin
    PUSH AX   ; preservo dir_retorno
    CMP DX, 0
    JNE prev
    PUSH ES:[BX+4]
    JMP Call
Prev:
    PUSH ES:[BX+2]
Call:
    PUSH DX
    CALL findNode
    POP BX
    POP AX
Fin:
    PUSH BX
    PUSH AX
    RET
findNode endp
```

Parte c)

Cada nodo ocupa 6 bytes: dos para el dato (de tipo short) y dos para los desplazamientos. Como sólo se utilizan los desplazamientos para identificar al siguiente nodo, entonces todos los nodos deben pertenecer al mismo segmento. Un segmento son 2^16 = 65536 bytes. La dirección 0 se utiliza para representar NULL. De esta forma, la cantidad máxima de nodos es el numero n que maximiza la siguiente ecuación: 6*n <= 65536 – 1. Entonces n = 10922.

Parte d)

El paso base consume 6 bytes de stack correspondientes a los 2 parámetros y la dirección de retorno.
En el llamado de la función verify, se utilizan 2k+4 bytes correspondientes a:
- 1 parámetro (dato)
- dirección de retorno

- 2k bytes que utiliza internamente la función verify
pero se debe tener en cuenta que al momento de llamar a verify se han recuperado 4 bytes de stack

El consumo neto para cada paso recursivo salvo el último es de 2 bytes correspondientes a la dirección de retorno.
Observar que verify y findNodo, no se llaman de forma anidada, por lo que para calcular el máximo uso del stack hay que considerar solamente la llamada a verify correspondiente al procesamiento del nodo n.
Por lo tanto el consumo total sera: 2*(n+k) + 6.

Parte e)

Para poder ejecutarse en cualquier caso debe cumplir, ademas de la condición calculada en la parte c, que el stack no desborde, es decir, que pueda contenerse en un solo segmento.

Es decir que
max(2*(n+k), 6*(n+1)) < 2^16 = 65536

Dependiendo del valor de k resulta cual de las dos restricciones es mayor.

Solución Ejercicio 2

Parte a)

```
// se mantiene una ventana de 10 segundos de revoluciones (10seg = 100tics -10hz-)
char rev_hist[100];
short i = 0, tics_5 = 5, tics_50 = 50,
char encendido = 0, freno_presionado = 0, freno_suelto = 0, luz  = 0, revs_per_timer =
0, total_revs = 0, estuvo_encendido = 0;

void main {
  //Instalar interrupciones
  for (int j = 0; j < 100; i++) rev_hist[j] = 0;
  out(CONTROL, 0);
  out(DISPLAY, 0);

  enable()

  while (true) {
    short status = in(STATUS);
    encendido = status & 0x0001;
    if (encendido) {
      char freno_estado_anterior = freno_presionado;
      freno_presionado = status & 0x0002;
      if (freno_presionado){
         freno_suelto = 0;
      } else if (freno_presionado == 0 && freno_estado_anterior != 0) {
        freno_suelto = 1;
        tics_50 = 1;
      }
      char potencia = (status & 0x01FC) >> 2;
      estuvo_encendido = 1;
      out(CONTROL, (potencia << 1) | luz);
    } else {
      out(DISPLAY, 0);
      out(CONTROL, 0);
      if (estuvo_encendido) {
        for (int j = 0; j < 100; i++) {
          rev_hist[j] = 0;
        }
        i = 0;
        tics_5 = 5;
        tics_50 = 50;
        freno_presionado = 0;
        freno_suelto = 0;
        luz = 0;
        revs_per_timer = 0;
        total_revs = 0;
        estuvo_encendido = 0;
      }
    }
  }
}

void interrupt hall() {
```

```
  if (encendido) {
    revs_per_timer++;
  }
}

void interrupt timer() {

  if (encendido) {
    total_revs -= rev_hist[i];
    total_revs += revs_per_timer;
    rev_hist[i] = revs_per_timer;
    i = (i + 1)%100;
    revs_per_timer = 0;
    if ((total_revs*6) > 99) {
      out(DISPLAY, 0x99);
    } else {
      char unidades = (total_revs*6 )% 10;
      char decenas = (total_revs*6 )/ 10;
      char disp_out = decenas << 4 | unidades
      out(DISPLAY, disp_out);
    }

    if (freno_presionado) {
      if (tics_5 == 0) {
        luz = luz ^ 0x01;  // (luz = luz == 0? 1 : 0);
        tics_5 = 5;
      }
      tics_5--;
    } else if (freno_suelto) { // titilar durante 5 segundos más
      if (tics_50 > 0) {
        if (tics_5 == 0) {
            luz = luz ^ 0x01;
            tics_5 = 5;
        }
        tics_5--;
      } else {
        freno_suelto = 0;
        luz = 0;
        tics_50 = 50;
      }
      tics_50--;
    }
  }
}
```

Parte b)

```
CONTROL EQU valor0
DISPLAY EQU valor1
STATUS EQU valor2
ID_INT_TIMER EQU valor3
ID_INT_HALL EQU valor4

.data
rev_hist db 100 dup 0
i dw 0
tics_5  dw 5
tics_50  dw 50
```

```
encendido db 0
freno_presionado db 0
freno_suelto db 0
luz db 0
revs_per_timer db 0
total_revs db 0
estuvo_encendido db 0

.code
main:

  XOR ES, ES
  MOV ES:[4*ID_INT_HALL+2], SEGMENT hall
  MOV ES:[4*ID_INT_HALL], OFFSET hall
  MOV ES:[4*ID_INT_TIMER+2], SEGMENT timer
  MOV ES:[4*ID_INT_TIMER], OFFSET timer
  OUT CONTROL, 0
  OUT DISPLAY, 0
  sti
```