Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2343 – Arquitectura de Computadores (II/2015)

Examen

Pregunta 1

a) Describa detalladamente que pasa si se intenta ejecutar un archivo ejecutable binario del computador básico, en un computador x86 de 16 bits. **(1 pto.)**
Solución: Puede pasar cualquier cosa. Ya que son códigos ejecutables para computadores distintos, lo más probable es que no tengan los mismos opcodes. Luego, al intentar ejecutar (si es posibel evitar la prohibición del sistema operativo), el computador tendrá un comportamiento no predecible.

b) Un *disassembler* es un programa que transforma código binario ejecutable en *assembly*. Describa como funcionaría un *disassembler* para el computador básico. ¿Es posible obtener el *assembly* original a partir de un programa ubicado en la memoria de instrucciones? **(2 ptos.)**
Solución: Un *disassembler* para el computador básico trabajaría tomando cada uno de los opcodes en las correspondientes instrucciones. Esto es posible gracias a que los opcodes tienen una estructura fija y que cada uno corresponde a sólo una instrucción del assembly. En el caso de las instrucciones que usan 2 opcodes, el disassembler deberá ser capaz de reconocer esta secuencia. Los literales son también sencillos de reconocer, dada la estructura de los opcodes. Finalmente, a partir de un programa ubicado en la memoria de instrucciones, no es posible reconstruir el codigo assembly original, ya que se pierde la información de los labels.

c) Describa detalladamente un mecanismo para generar un archivo ejecutable en x86, a partir del resultado de un *assembler* para el computador básico. Describa claramente como trabaja cada una de las transformaciones intermedias. **(3 ptos.)**
Solución: El primer paso consiste en generar el assembly del computador básico usando el *disassembler* a partir del código ejecutable almacenado en la memoria de instrucciones. Luego, a partir de este assembly, se genera assembly x86, traduciendo cada instrucción a su simil en x86. Dado que el assembly corresponde a un computador de 16 bits, no hay problema en usar la siguiente conversión: el registro A se transforma en AL y el registro B en BL. Para poder utilizar las variables declaradas en la memoria de datos, se copia el contenido de esta en el código assembly x86, después de las instrucciones. De esta manera, cada vez que en el assembly del computador básico se referencia al contenido de la memoria de datos, se debe transformar esta dirección a la nueva ubicación, de acuerdo a lo expuesto anteriormente. Finalmente, el código assembly x86 se procesa con un *assemler* x86, para generar el código ejecutable.

Pregunta 2

a) Un computador x86 monoprocesador posee un pipeline de 5 etapas que se ejecutan en el siguiente orden:

   a) Fetch: Obtiene desde la memoria el opcode de la instrucción a ejecutar.

   b) Decode: Decodifica el opcode, enviando las señales correspondientes a cada componente.

   c) Read: Lee desde registros y memoria los datos requeridos para ejecutar la operación.

   d) Execute: Ejecuta la operación aritmética/lógica de la instrucción usando la ALU.

   e) Write: Almacena en registros o memoria el resultado de la operación aritmética/lógica.

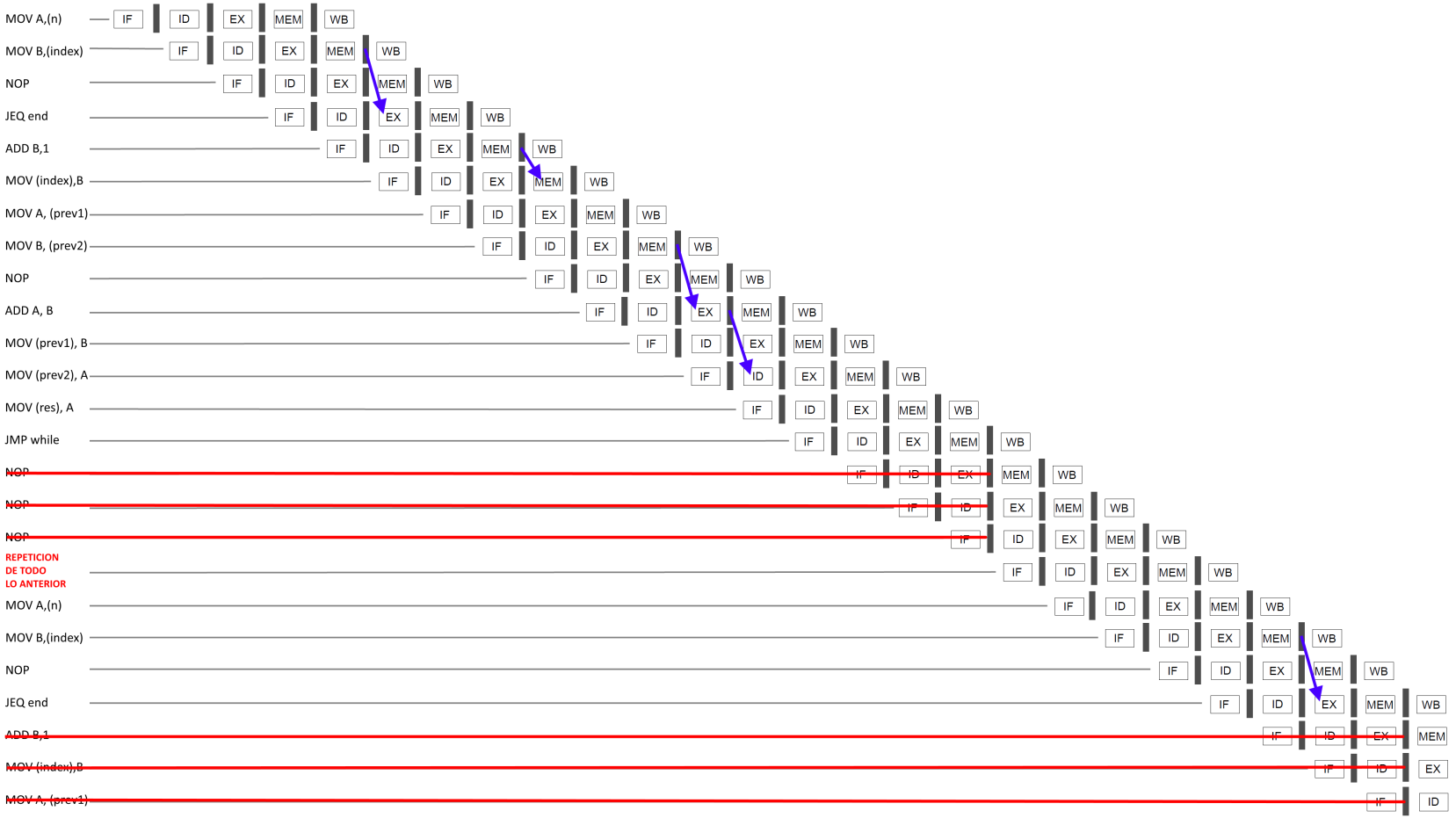
   Además de los mecanismos tradicionales para combatir hazards de datos y de control, el computador evita hazards estructurales de acceso a memoria entre las etapas Fetch, Read y Write, al utilizar una memoria RAM que permite realizar de manera simultánea tres solicitudes de distintas. A pesar de esto, es posible generar un hazard estructural de ejecución entre las etapas Read y Execute, cuando se procesa una instrucción del tipo MOV Reg1, [Reg2+offset]. ¿Cómo es posible evitar este hazard ? **(2 ptos.)**
   Solución: El hazard se genera al existir una operación artimética para calcular el direccionamiento. Esto implica que se utiliza la ALU en dos etapas. Para solucionar el hazard, se puede agregar un sumador en la etapa encargada de ir a la memoria, de manera que el direccionamiento sea calculado sin generar colisiones con otras etapas.

b) Determine el número de ciclos que se demora el siguiente código, detallando en un diagrama los estados del pipeline por instrucción. El pipeline tiene forwarding entre todas sus etapas, el manejo de stalling es por software (instrucción NOP) y predicción de salto asumiendo que no ocurre. Indique en el diagrama cuando ocurre forwarding, stalling y flushing. **(4 ptos.)**

```
DATA
  n     2
  index 0
  prev1 0
  prev2 1
  res   1

CODE
  while:
    MOV A, (n)
    MOV B, (index)
    JEQ end
    ADD B, 1
    MOV (index), B
    MOV A, (prev1)
    MOV B, (prev2)
    ADD A, B
    MOV (prev1), B
    MOV (prev2), A
    MOV (res), A
    JMP while
  end:
```

Solución: El programa toma 46 ciclos.


MOV A,(n)
MOV B,(index)
NOP
JEQ end
ADD B,1
MOV (index),B
MOV A, (prev1)
MOV B, (prev2)
NOP
ADD A, B
MOV (prev1), B
MOV (prev2), A
MOV (res), A
JMP while
NOP
NOP
NOP
REPETICION
DE TODO
LO ANTERIOR
MOV A,(n)
MOV B,(index)
NOP
JEQ end
ADD B,1
MOV (index),B
MOV A, (prev1)
IF
ID
EX
MEM
WB

Pregunta 3

a) ¿En qué se diferencian los tipos de paralelismo SIMD y SIMT? Complemente las diferencias con ejemplos para cada uno de los casos. **(1 pto.)**
Solución: SIMD (Single Instruction Multiple Data) consiste en utilizar un mismo programa (o función) para una serie de de datos. Un ejemplo de esto son las instrucciones SSE de la ISA x86. SIMT (Single Instruction Multiple Threads) consiste en utilizar un mismo programa sobre múltiples datos, pero utilizando múltiples threads. Un ejemplo de esto son las GPUs.

b) Asuma que cuenta con un computador x86 de 16 bits con soporte para instrucciones SSE de 64 bits. Cada uno de los 8 registros extra, XMM0 a XMM7, puede ser utilizado también como 4 registros independientes de 16 bits, *i.e.*, los subregistros de XMM0 son XMM0_0, XMM0_1, XMM0_2 y XMM0_3. Las instrucciones aritméticas que permiten utiliazar estos registros son MULPS Reg0, Reg1 y ADDPS Reg0, Reg1, mientras que la instrucción MOV es utilizada para las transferencias. Tomando todo esto en consideración, implemente en assembly x86 de 16 bits + SSE64, un programa que calcule el producto punto entre 2 vectores de enteros de 16 bits de largo arbitrario N. **(2 ptos.)**
Solución: Se asume que los vectores están almacenados en A y B.

```
    JMP    main
    N      dw    ?
    A      dw    ?,?,?,...
    B      dw    ?,?,?,...
    res    dw    0
  main:
    MOV    SI, 4
    MOV    DI, 8
    MOV    XMM2_0, 0
    MOV    XMM2_1, 0
    MOV    XMM2_2, 0
    MOV    XMM2_3, 0
    LEA    BX, A
    LEA    CX, B
  for:
    CMP    SI, N
    JG    resto
    MOV    XMM0_0, [BX+DI-8]
    MOV    XMM0_1, [BX+DI-6]
    MOV    XMM0_2, [BX+DI-4]
    MOV    XMM0_3, [BX+DI-2]
    MOV    XMM1_0, [CX+DI-8]
    MOV    XMM1_1, [CX+DI-6]
    MOV    XMM1_2, [CX+DI-4]
    MOV    XMM1_3, [CX+DI-2]
    MULPS  XMM0, XMM1
    ADDPS  XMM2, XMM0
    ADD    SI, 4
    ADD    DI, 8
    JMP    for
  resto:
    SUB    SI, 4
    SUB    DI, 8
  for_resto
    CMP    SI, N
    JEQ    end
```

```
    MOV    AX, [BX+DI]
    MUL    [CX+DI]
    ADD    res, AX
    ADD    SI, 1
    ADD    DI, 2
    JMP    for_resto
  end
    ADD    res, XMM2_0
    ADD    res, XMM2_1
    ADD    res, XMM2_2
    ADD    res, XMM2_3
```

c) Se desea programar en una GPU un algoritmo que procese una imagen en escala de grises, guardada por filas, sustituyendo el valor de cada pixel por el promedio de los valores de sus 8 vecinos.

I. ¿Cuál sería la tarea de cada uno de los threads de este algoritmo? **(1 pto.)**
Solución: Cada thread se encargaría de calcular el valor de un pixel, *i.e.*, calcular el promedio de los 8 vecinos.

II. ¿Cómo definiría un *warp* para este algoritmo? **(1 pto.)**
Solución: Dado que la matriz está guardada por filas, un petición de memoria no alcanza a traer los datos de los 8 vecinos. En promedio, se necesitarán 3 peticiones distintas para los 8 vecinos (datos en 3 filas distintas). Por lo tanto, tun warp debería definirse como un grupo de threads que procesen pixeles en la misma fila y que las peticiones de lecturas del primer thread del warp traigan también a la caché L1 todos los datos que necesitan los otros threads del warp

III. ¿Cómo se podrían evitar los problemas de divergencia de código en este algoritmo? **(1 pto.)**
Solución: Una posibilidad para evitar la divergencia (control de flujo) es tener 5 funciones distintas para procesar los pixeles de los 4 bordes y los del que no estan en los bordes.