Universidad de San Andrés
I301 Arquitectura de computadoras y Sistemas Operativos

Profesor: **Daniel Veiga**

Ayudantes: **Nicolás Romero, Juan Carlos Suárez, Tomás Agustín Chimenti, Joaquín Torres, Tobías Moraut, Santiago Pierini**

Entrega: **27/03/2025 11:59 PM**

Entrega tarde (**máxima nota 8**): **Domingo 30/03/2025 11:59 PM**

Repositorio:

https://github.com/SPI-Udesa/TP_ACSO.git

El código del trabajo práctico está en el repositorio, incluye las cosas descritas en este documento.

Introduccion

Este trabajo práctico se basa en un ejercicio práctico propuesto en el curso "Computer Architecture" de la Universidad de Chicago.

El objetivo es desarrollar un programa en C que simule la ejecución de algunas instrucciones del conjunto ARMv8. Este simulador modelará el comportamiento de cada instrucción, permitiendo correr programas escritos para ARMv8 y observar sus resultados. En esencia, el programa actúa como un simulador de CPU, replicando el proceso que sigue el procesador al leer y ejecutar instrucciones en lenguaje ensamblador.

El simulador debe cargarse con un archivo de entrada que contenga un programa en ARMv8. El formato de archivo de input es que cada línea contiene una instrucción escrita en formato hexadecimal. Por ejemplo **0xea020020** representa la instrucción **ANDS X0, X1, X2**. El simulador debe ejecutar una instrucción a la vez, después de ejecutar cada una modifica el estado del CPU, esto incluye los registros, la memoria, y las flags. Para generar las instrucciones hexadecimal, se escriben en assembly, y corren un ensamblador que produce las instrucciones en "byte code" o hexadecimal de 32 bits.

El Simulador está dividido en dos secciones.

- El shell
- El simulador

El objetivo de este trabajo práctico es implementar el simulador. En el directorio **/src** van a encontrar dos archivos **shell.c** y **shell.h** que implementan el shell. El tercer archivo **sim.c** es donde deben implementar el simulador. Siéntanse libres de usar otros archivos y agregarlos al Makefile.
No deben cambiar el código en shell.c/.h

El Shell

El propósito de el shell es proveer al usuario comandos que controlan la ejecución del simulador. Tiene los siguientes comandos:

1. **go**: esto corre el programa hasta que el simulador indique halt (HLT).
2. **run <n>**: simula la ejecución de **n** instrucciones.
3. **mdump <low> <high>**: Imprime el contenido de la memoria desde la dirección <low> a la dirección <high> a la pantalla y al archivo que contiene el dump (**dumpsim**). La memoria empieza en 0x10000000, y tiene un tamaño de 0x00100000. **Miren el codigo en shell.c**
4. **rdump**: Imprime el contenido de los registros X0-X31, Flags N, Z, el registro PC y la cantidad de instrucciones ejecutadas.
5. **input reg_num reg_val**: importa el valor de **reg_val** al registro **reg_num.**
6. **?:** muestras estas instrucciones.
7. **quit**: sale del shell.

Para ejecutar el shell hay que usar uno o más argumentos de línea de comando como:

```
dir_de_sim$ ./sim ands.x
```

El shell va a cargar el programa en la memoria simulada, y crea un archivo **dumpsim** para guardar información útil de la ejecución.

El Simulador

Como mencionamos arriba, el simulador simula cada instrucción del programa ARM. Lo hace leyendo de memoria, donde es cargado por el shell cuando arranca el programa. Durante la ejecución de las instrucciones debe modificar el estado del CPU de acuerdo con las descripción del ISA de las instrucciones en el manual de referencia de ARMv8 incluido en el directorio **ref/**. La última sección del TP explicita qué instrucciones se tienen que implementar.

El estado del CPU incluye el PC, los 32 registros de uso general, 2 flags, y la memoria. Está definido en la variable (en **shell.h)**:

```
#define ARM_REGS 32

typedef struct CPU_State {
  uint64_t PC;              /* program counter */
  int64_t REGS[ARM_REGS];   /* register file. */
  int FLAG_N;               /* flag N */
  int FLAG_Z;               /* flag Z */
} CPU_State;

CPU_State STATE_CURRENT, STATE_NEXT;
int RUN_BIT;
```

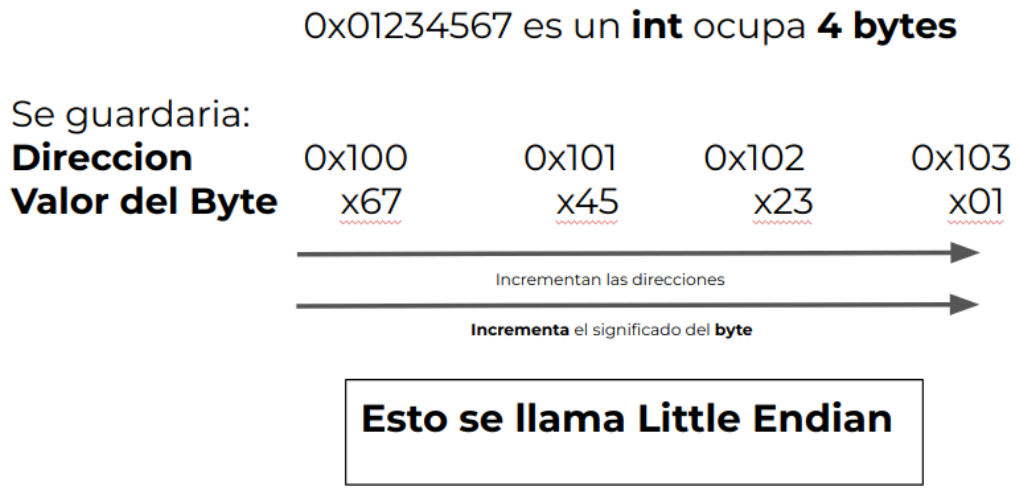
Solamente estamos implementando las banderas N y Z las otras dos (Carry [C] y Overflow [V] pueden asumir que siempre son **0**)

Para acceder a la memoria deben usar las siguientes funciones, provistas en **shell.c/h**

```
uint32_t mem_read_32(uint64_t address);
void mem_write_32(uint64_t address, uint32_t value);
```

Memoria

Como explicamos en clase, la memoria de ARM es little-endian, y direcciona a la mínima unidad de un byte. Little endian significa que escribe el byte menos significativo en la dirección más baja y el byte más significativo en la dirección más alta.



Y 64 bits se escriben asi:

```
long lorder = 0x1211341156117811;
// 0x100 0x101 0x102 0x103 0x104 0x105 0x106 0x107
//  11    78    11    56    11    34    11    12
```

Para hacer Load y Store (LDUR/H/B y STUR/H/B), necesitan **tomar en cuenta como está y cómo va a quedar** la memoria ya que solamente pueden acceder a través de la función que trae y guarda 32 bits.

Estado del CPU

En la función **cycle()** que se encuentra en shell.c ejecutamos los siguiente:

```
CURRENT_STATE = NEXT_STATE
```

Por lo tanto todos sus cambios deberían de hacerse en **NEXT_STATE**. **INSTRUCTION_COUNT** cuenta cuantas instrucciones se ejecutaron, tambiénse actualiza en **cycle()** así que no tienen que implementar esta lógica.

El shell llama a la función **process_instruction()** la cual ustedes deben implementar en **sim.c**.

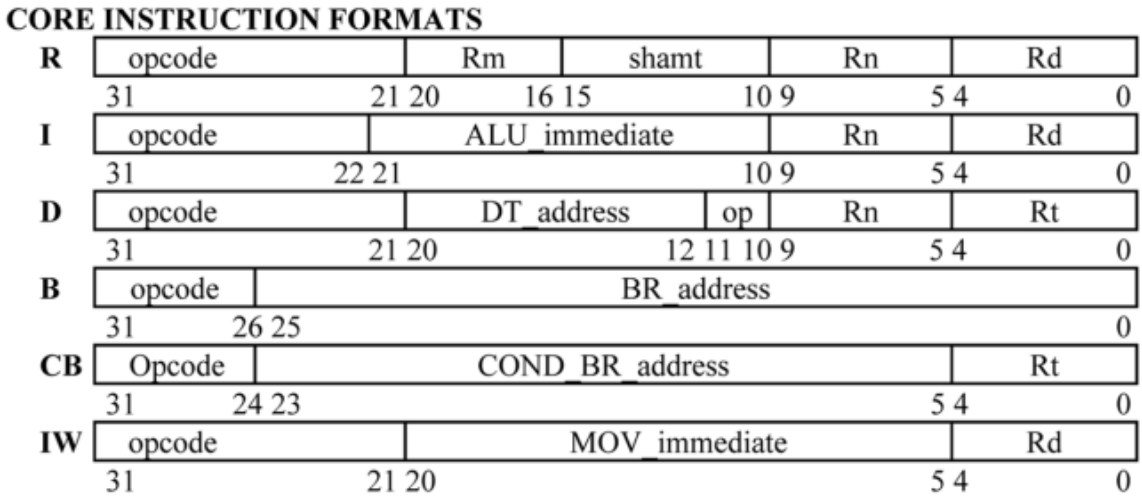
La Tarea

El archivo **sim.c** incluye la función vacía **process_instruction()** que es llamada desde el shell por cada instrucción de assembly a ejecutar. Su responsabilidad es implementar esta función y simular las instrucciones. Pueden agregar estructuras, funciones y archivos que los ayuden a hacer el trabajo. Usen algo de tiempo para organizar el código, así es fácil de leer y prolijo.

Hay dos partes claves al ejecutar una instrucción:

1. Primero hay que buscar la instrucción y decodificarla. Esto involucra entender qué instrucción es, e interpretar los componentes de la instrucción. En general esto se llama el **decode** stage. Sería valioso crear una estructura genérica que pueda guardar los componentes de las instrucciones.
2. Luego está la ejecución de la instrucción con sus componentes y asegurándose de actualizar el estado del CPU. En general esto se llama el **execute** stage.

Las instrucciones a implementar pueden parecer numerosas, pero se agrupan en categorías, y dentro de cada categoría, presentan pequeñas variaciones. A continuación, se presenta un esquema preliminar de estas categorías. Sin embargo, es fundamental verificar cada instrucción en el manual de ARMv8 para asegurar su correcta implementación. Además, es importante revisar las especificaciones detalladas al final del documento..



Primero se corregirá la funcionalidad, que debe ser la prioridad principal del TP. **En particular, que el estado del CPU sea correcto al finalizar la ejecución de cada instrucción, esto incluye Registros, Flags, PC, Memoria, etc.** Para verificar la implementación de cada instrucción, probaremos su simulador con varios programas, algunos proporcionados por nosotros y otros no, para asegurarnos de que todas las instrucciones se ejecuten correctamente.

Simulador de referencia (./ref_sim)

El repositorio contiene ./**ref_sim** que es el simulador que vamos a usar para corregir el práctico. Su simulador tiene que producir el mismo estado del CPU después de cada instrucción que el simulador de referencia. Vamos a usar el simulador de referencia para corregir el TP.

Corran los programas que están en el directorio **/input** y armen sus propios programas usando el **README** para compilar de assembly a hex. Les recomendamos escribir un test para cada instrucción, y hacer **run 1** y verificar el valor de los registros con **rdump** y la memoria con **mdump.**

Archivos en el Repositorio

/

Contiene directorios mencionados abajo y dos archivos:
-**ref_sim**: es un simulador que es de referencia para ver que debería de hacer el suyo, se usa de la misma manera.
-**README.md**: Instrucciones de que incluye el repositorio y como usarlo.

/src

Contiene los archivos de shell.c y shell.h que no se deben modificar. Contiene sim.c donde va su código. Contiene **Makefile** para compilar el programa.

```
tp1-acso$ cd src
src$ make
gcc -g -O0 shell.c sim.c -o sim
#remueve la compilacion
src$ make clean
rm -rf *.o *~ sim
```

Si quieren agregar archivos .c acuerdense de modificar el makefile.

/inputs

Esto contiene los archivos .s que tienen assembly ARMv8, acá pueden hacer más tests y usar asm2hex como describe el README.md para generar el byte code que se usa en el simulador.

/ref

Contiene el manual del ISA de ARMv8. Algunos tips que dijimos en clase:
-Registros: Pag. 71
-Instrucciones: Pág. 521-973

-Condition codes: Pag. 138-139

/aarch64-linux-android-4.9

Parte del toolchain del compilador de Google Android, usado por asm2hex en /inputs para generar el bytecode.

Entrega:

Envíen por email el enlace al repositorio que utilizarán para la entrega. El repositorio debe tener un directorio llamado `src` en la raíz, donde se encuentren los archivos necesarios para compilar el simulador. Si van a entregar fuera de plazo, envíen un correo antes de la fecha límite informándonos del retraso, para que no corrijamos el contenido que esté en el repositorio.

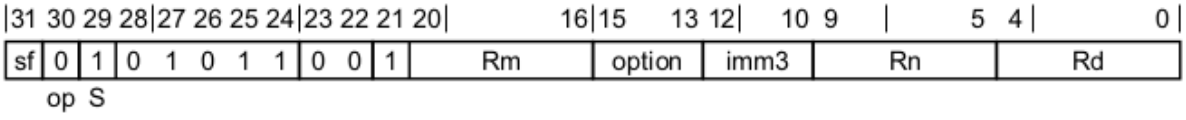
Consejos / recordatorios importantes:

- Escriban Tests para cada instrucción, así tienen confianza en su simulador.
  Hay instrucciones para compilar nuevos tests puestos en **/inputs** en el README
- Modifiquen solamente **sim.c**, si agregan archivos modifiquen en Makefile
- Los resultados del simulador deben coincidir con el **ref_sim** en cada instrucción ejecutada. Usen **rdump** y **mdump** después de cada instrucción para verificar que así sea.

Instrucciones a Implementar (sólo las de 64 bits hay que implementar):

Aclaración de Especificaciones:

1. Repetimos, **solo** tienen que hacer la variante de **la instrucción para 64 bits.**
2. **ADDS (Extended Register)** tiene el siguiente formato en el manual:



   Pero cuando compilamos la instrucción **ADDS Xd, Xn, Xm** el bit 21 de output es 0, no 1. Por favor asegúrense de que ande **ADDS**. Lo mismo ocurre con **SUBS.**

3. Cuando hacemos **ADDS (Extended Register)** vemos eso:

   64-bit variant

   Applies when sf == 1.

   ```
   ADDS <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
   ```

   No tienen que implementar la parte de extended y amount (O sea, pueden asumir que es 0). Solo implementen **ADDS Xd, Xn, Xm**. Lo mismo con **SUBS.**
4. Para **ADDS (Immediate)**

   Decode for all variants of this encoding

   ```
   integer d = UInt(Rd);
   integer n = UInt(Rn);
   integer datasize = if sf == '1' then 64 else 32;
   bits(datasize) imm;

   case shift of
       when '00' imm = ZeroExtend(imm12, datasize);
       when '01' imm = ZeroExtend(imm12:Zeros(12), datasize);
       when '1x' ReservedValue();
   ```

   No hay que implementar Reserved Value, pero si los otros dos. Lo mismo con **SUBS,** (y extra credit, add)
5. Las instrucciones LSL y LSR, y como mencionamos en el punto 4 ADDS y SUBS usan shift. Para todas las demás instrucciones, pueden asumir que **el shift (o shift amount [shamt]) es cero.**
6. Cuando tienen que actualizar los flags, como solo implementamos **Zero Flag (Z)** y **Negative Flag (N)**, y no implementamos **Carry(C)** y **Overflow(V)** pueden asumir que estas últimas son cero. Esto es particularmente relevante para **B.cond**. **Flag Z** se prende cuando el resultado de la operación aritmética es cero, y **Flag N** se prende cuando el resultado es negativo.
7. **X31** deben considerarlo el registro **XZR** es un registro que siempre tiene ceros. Esto es relevante para **CMP**
8. Las instrucciones de Load y Store **funcionan a partir de la dirección 0x10000000,** y tienen un tamaño de 0x00100000, que quiere decir que se pueden hacer Stores y Loads a partir de:
   Stur X0, **[0x10000000, 0x0]** hasta Stur X0, **[0x10000000, 0x00100000].** Si miran el código de shell.c van a ver por qué, pero no tienen nada que ver con la implementación de sim.c. Para testear estas instrucciones, vean como hicimos el **archivo sturb.s**

Instrucciones a implementar

ADDS (Extended Register, Immediate):
Immediate: **adds X0, X1, 3** (descripción: X0 = X1 + 3, luego updatear flags)
El caso de shift == 01 se debe implementar, osea moviendo el imm12, 12 bits a la izquierda. Tambien shift 00, pero no el ReservedValue

Extended Register: **adds X0 = X1, X2** (descripción: X0 = X1 + X2, luego updatear flags)

SUBS (Extended Register, Immediate)
Immediate: **subs X0, X1, 3** (descripción: X0 = X1 - 3, luego updatear flags)
El caso de shift == 01 se debe implementar, osea moviendo el imm12, 12 bits a la izquierda. .

Extended Register: **subs X0 = X1, X2** (descripción: X0 = X1 - X2, luego updatear flags)

HLT
hlt 0 (descripción: detener la simulación)
Opcode: 0x6a2
Esta instrucción simplemente debería que el comando **go** pare la simulación y vuelva al shell. Básicamente, setear la variable global RUN_BIT a 0.

CMP (Extended Register, Immediate)
Extended register: **cmp X13, X14** (descripción: XZR = X13 - X14, luego updatear flags). XZR es el registro cero, así que se calcula el resultado y se updatean las flags, pero se tira el resultado.

Immediate **cmp X13, 4** (descripción: XZR = X13 - X14, luego updatear flags)

ANDS (Shifted Register)
ands X0, X1, X2 (descripción: X0 = X1 & X2, luego updatear flags)
En el opcode se considerar que shift y N son siempre ceros, por lo que se chequean los bits <31:21>
No se tiene que implementar el shift

EOR (Shifted Register)
eor X0, X1, X2 (descripción: X0 = X1 ^ X2)
En el opcode se considerar que shift y N son siempre ceros, por lo que se chequean los bits <31:21>
No se tiene que implementar el shift

ORR (Shifted Register)
orr X0, X1, X2 (descripción: X0 = X1 | X2)
En el opcode se considerar que shift y N son siempre ceros, por lo que se chequean los bits <31:21>

No se tiene que implementar el shift

B
b target:
.
.
target:
(descripción: saltar a la instrucción de target, el target se calcula relativo a donde está apuntando el PC, así que puede ser positivo o negativo el salto, prestar especial atención. **Imm26:'00** quiere decir, el immediate seguido de dos bits siendo cero, osea un numero de 28 bits.)

BR
br X1 (descripción: saltar a la dirección guardada en el registro X1)

Debajo hay distintos casos de B.Cond,
Todas tienen el mismo opcode, se diferencian con el campo de **cond** los distintos casos se encuentran en la página 138-139

BEQ (B.Cond)
cmp X1,X2
beq target
.
.
target
(descripción: salto a target si X1 == X2, se valida el caso con los flags. Si requiere flags C o V, asumir que son cero. Vale para todos los b.conditional. Esta instrucción es un caso de b.cond)

BNE (B.Cond)
cmp X1,X2
bne target
.
.
target
(descripción: salto a target si X1 != X2, se valida el caso con los flags. Si requiere flags C o V, asumir que son cero. Vale para todos los b.conditional. Esta instrucción es un caso de b.cond)

BGT (B.Cond)
cmp X1,X2
bgt target
.
.

target
(descripción: salto a target si X1 > X2, se valida el caso con los flags. Si requiere flags C o V, asumir que son cero. Vale para todos los b.conditional. Esta instrucción es un caso de b.cond)

BLT (B.Cond)
cmp X1,X2
blt target
.
.
target
(descripción: salto a target si X1 < X2, se valida el caso con los flags. Si requiere flags C o V, asumir que son cero. Vale para todos los b.conditional. Esta instrucción es un caso de b.cond)

BGE (B.Cond)
cmp X1,X2
bge target
.
.
target
(descripción: salto a target si X1 >= X2, se valida el caso con los flags. Si requiere flags C o V, asumir que son cero. Vale para todos los b.conditional. Esta instrucción es un caso de b.cond)

BLE (B.Cond)
cmp X1,X2
ble target
.
.
target
(descripción: salto a target si X1 <= X2, se valida el caso con los flags. Si requiere flags C o V, asumir que son cero. Vale para todos los b.conditional. Esta instrucción es un caso de b.cond)

LSL (Immediate)
lsl X4, X3, 4 (descripción: Logical left shift (X4 = X3 << 4))

LSR (Immediate)
lsr X4, X3, 4 (descripción: Logical right shift (X4 = X3 >> 4))

STUR -
stur X1, [X2, #0x10] (descripción: M[X2 + 0x10] = X1)
Acuerdense que en el simulador la memoria empieza en 0x10000000, ver especificaciones, no cambia la implementación pero si el testeo.

STURB

sturb X1, [X2, #0x10] (descripción: M[X2 + 0x10](7:0) = X1(7:0), osea los primeros 8 bits del registro son guardados en los primeros 8 bits guardados en la dirección de memoria). Importante acordarse que la memoria es little endian en Arm.
Acuerdense que en el simulador la memoria empieza en 0x10000000, ver especificaciones, no cambia la implementación pero si el testeo.

STURH
sturh W1, [X2, #0x10] (descripción: M[X2 + 0x10](15:0) = X1(15:0), osea los primeros 16 bits del registro son guardados en los primeros 16 bits guardados en la dirección de memoria). Importante acordarse que la memoria es little endian en Arm.
Acuerdense que en el simulador la memoria empieza en 0x10000000, ver especificaciones, no cambia la implementación pero si el testeo.

LDUR
ldur X1, [X2, #0x10] (descripción: X1 = M[X2 + 0x10])
Acuerdense que en el simulador la memoria empieza en 0x10000000, ver especificaciones, no cambia la implementación pero si el testeo.

LDURH
ldurh W1, [X2, #0x10] (descripción: X1= 48'b0, M[X2 + 0x10](15:0), osea 48 ceros y los primeros 16 bits guardados en la dirección de memoria)
Acuerdense que en el simulador la memoria empieza en 0x10000000, ver especificaciones, no cambia la implementación pero si el testeo.

LDURB
ldurb W1, [X2, #0x10] (descripción: X1= 56'b0, M[X2 + 0x10](7:0), osea 56 ceros y los primeros 8 bits guardados en la dirección de memoria)
Acuerdense que en el simulador la memoria empieza en 0x10000000, ver especificaciones, no cambia la implementación pero si el testeo.

MOVZ
movz X1, 10 (descripción: X1 = 10, Solo hay que implementar la condición donde hw = 0, osea shift es cero.)

ADD (Extended Register & Immediate)
Immediate: add X0, X1, 3 (descripción: X0 = X1 + 3)
El caso de shift == 01 se debe implementar, osea moviendo el imm12, 12 bits a la izquierda. También se debe implementar shift 00, pero no el caso de ReservedValue.

Extended Register: add X0 = X1, X2 (descripción: X0 = X1 + X2)

MUL
mul X0, X1, X2 (descripción X0 = X1 * X2)

CBZ
cbz X3, label
.
.
label (descripción: saltar a label, si X3 es 0)

CBNZ
cbnz X3, label
.
.
label (descripción: saltar a label, si X3 no es 0)