Trabajo Práctico

Diseño de procesadores

Materia: Programación de softcores en FPGAs
Programa de Profesoras/es Visitantes

Material disponible en: `https://github.com/fokerman/fpgaSoftcoreProgrammingCourse`

1. Introducción

El objetivo de este trabajo práctico es diseñar la arquitectura de un procesador, con sus instrucciones, lenguaje ensamblador, codificaciones y características propias. Para luego implementar dos posibles organizaciones que respeten la arquitectura diseñada. El procesador en cualquiera de sus dos organizaciones, deben ser capaz de ejecutar programas en lenguaje ensamblador. En este trabajo se pedirá implementar un juego tipo Simon.

2. Enunciado

El enunciado consta de cinco partes, la primera invita a diseñar la arquitectura de un procesador simple, similar a un modelo Von Neumann clásico. La segunda y tercera parte desarrollan este procesador implementadolo con dos *datapaths* diferentes, uno de un solo bus y una organización microprogramada, mientras que el segundo construye un modelo donde cada instrucción se ejecuta en un ciclo de reloj. La cuarta parte consiste en implementar el código de un juego para los dos procesadores desarrollados. La última parte consiste en extender el soporte de la arquitectura para alguna característica particular.

En este enunciado no hay necesariamente pasos a seguir, sino referencias de todo lo que debe soportar y poder realizar el procesador que se desarrolle. Esto implica que puede ser necesario revisitar la definición de la arquitectura en base a una alteración de la organización. Sobre todo para hacer más simple la organización del procesador, en especial para el *datapath* de un solo ciclo y el último ejercicio.

El procesador que se espera que diseñen en este trabajo debe respetar las siguientes características:

2.1. Características mínimas

A continuación se lista un conjunto de características que debe respetar su diseño. En caso que lo consideren necesario puede alterar alguna de estas características, siempre que su alteración tenga mejor o mayor soporte que el original.

- Tamaño de palabra, operaciones y datos de 8 bits. Instrucciones de tamaño fijo de 16 bits.
- Registros expuestos al programador, al menos 6 registros de propósito general, un registro para el PC y un registro de Flags, Carry, Negative, Overflow, Zero.
- Modos de direccionamiento. Se debe poder soportar, dependiendo de cada instrucción, los modos de direccionamiento: inmediato, directo a registro, directo a memoria, indirecto a registro.
- Instrucciones. Mínimamente se deben soportar las siguientes instrucciones:
  - Aritméticas y lógicas: Addition, Subtraction, Logic Or, Logic And, Logic Not, Compare, Logical Shift Right, Logical Shift Left. Estas deben operar mínimamente con cualquier registro de propósito general.
  - CopyRegister: Instrucción para copiar el contenido de un registro en otro.

    - Saltos: Jump, Jump Equal, Jump Not Equal
    - Subrutinas: callSubrutine, ReturnSubrutine. Instrucciones para llamar a subrutinas, estas instrucciones se pueden implementar utilizando la pila o utilizando registros específicos donde almacenar la dirección de retorno.
    - GetFlags: Instrucción que permita leer los Flags, ya sea escribiéndolos en un registro o en memoria.
    - Memoria: writeToMem, readFormMem. Instrucciones para leer y escribir en memoria.
    - SetRegister: Instrucción para asignar un valor inmediato a un registro.
- Memoria de datos: 1kB, direcciones de 10 bits con direccionamiento a 8 bits.
- Memoria de código: 1kB, direcciones de 9 bits con direccionamiento a 16bits.
- Pila: Puede ser implementada de forma estática en una estructura no acessible por el programador o dinámica en memoria. En cualquier caso debe soportar al menos 5 llamados a funciones anidadas.
- Entrada/salida: El procesador dispondrá de un puerto de 4 bits de entrada y un puerto de 4 bits de salida. Ambos puertos deben estar mapeados a memoria en alguna dirección fija designada.

Observaciones

La memoria de código y de datos tienen diferentes unidades direccionables, dos unidades direccionables de datos se requieren para tener una unidad direccionable de la memoria de código. Si bien son del mismo tamaño, una debe tener el doble de direcciones que la otra. Esto implica que se debe tener especial cuidado cuando se considera una dirección en la memoria de datos y en la memoria de instrucciones.

Notar que en ningún lugar se indica si la memoria de datos y código debe ser compartida. Se recomienda diseñar la arquitectura considerando que las memorias son independientes, sin embargo tienen esta libertad.

Las características mencionadas son las mínimas que debe cumplir la arquitectura. Es posible agregar más instrucciones, o más registros o más puertos de entrada salida o incluso soporte para interrupciones.

2.2. Ejercicio 1: Diseño de arquitectura

Utilizando como base las características enunciadas en el punto anterior, se pide diseñar una arquitectura de un procesador que las respete.

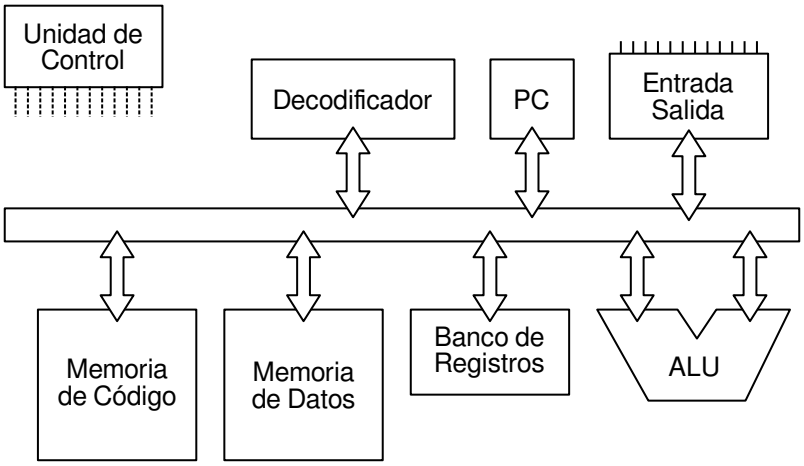
- Proponer un nombre a su arquitectura.
- Definir la arquitectura del procesador, documentar y describir cada una de sus características principales.
- Definir el conjunto de instrucciones soportadas por el procesador, indicando el comportamiento de cada una.
- Definir una codificación para cada una de las instrucciones, indicar los campos y el comportamiento que debe realizar el decodificador de instrucciones para tomar cada uno de los campos.
- Definir un lenguaje ensamblador, indicar lo nemónicos de cada instrucción y como se indican los parámetros. El lenguaje debe incluir palabras reservas para definir constantes en memoria y resolver etiquetas.
- Desarrollar un programa que tome un archivo de texto y construya un archivo ensamblado. El programa debe retornar otro archivo de texto codificando los bytes en caracteres hexadecimales. Este archivo se utilizará luego como entrada de la memoria de código. Se recomienda realizar el desarrollo en `Python`.

- Implementar tres programas de ejemplo:
  - Un programa que realice un ciclo escribiendo los números del 1 al 10 en la memoria de datos.
  - Un programa que lea datos de memoria previamente escritos y cuente todas las apariciones de un determinado valor `target` en un rango de memoria dado.
  - Un programa que lea los 4 bits del puerto de entrada, interprete el valor como un numero entero, calcule el inverso aditivo y escriba el resultado en el puerto de salida.

El resultado de este ejercicio debe ser una documentación detallada del diseño de la arquitectura. Equivalente a una hoja de especificaciones o manual.

2.3. Ejercicio 2: Organización microprogramada

Llamaremos una organización microprogramada a una implementación clásica de un modelo de computadora con un solo bus compartido, con memorias independientes para código y datos. La unidad de control se ocupará de activar secuencialmente las señales para realizar el ciclo de instrucción y resolver cada una de las instrucciones.

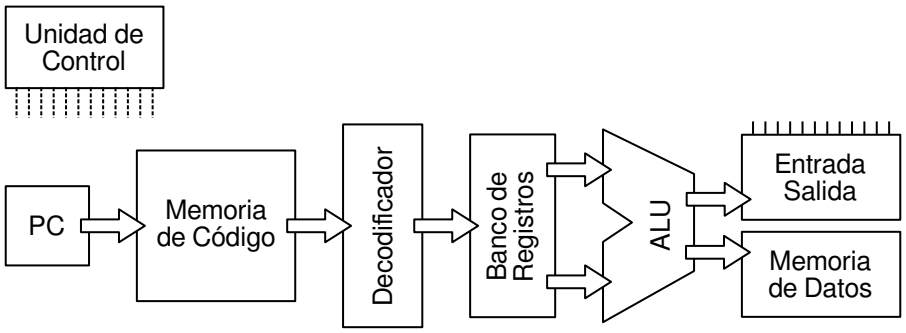


- Diseño del *datapath*, identificar todos los registros temporales que se utilizarán y que información se almacenará en cada registro.
- Definir el ciclo de instrucciones, como se leerá una instrucción y los pasos para que sea decodificada para luego ser ejecutada.
- Definir las señales de control del *datapath*, para esto recorrer instrucción por instrucción caracterizando las señales que serán utilizadas en cada flujo de control.
- Diseñar la secuencia de activación de señales para resolver cada instrucción. Se recomienda utilizar una memoria dentro de la unidad de control para almacenar las codificaciones. Si son muchas señales es recomendable implementar un programa ayude a construir la entrada para la memoria.
- Implementar en `Verilog` el componente ALU, realizar un *testbech* de todas las operaciones que debe soportar.
- Implementar en `Verilog` el componente PC, realizar un *testbech* que pruebe como es incrementado y los valores que toma.
- Implementar en `Verilog` el banco de registros, realizar un *testbech* de la escritura y lectura de registros.
- Implementar en `Verilog` los módulos de memoria, tanto de código como de datos o los dos simultáneamente. Considerar que la memoria debe tener en cuenta las señales para controlar los puertos de entrada/salida mapeados a memoria. Realizar un *testbech* que pruebe escrituras y lecturas en memoria, teniendo en cuenta los puertos mapeados.

- Implementar en `Verilog` el módulo que permite exponer o recibir datos de entrada/salida. Debe responder a las mismas señales que la memoria, ya que está mapeado a memoria. Realizar un *testbech* que pruebe leer y escribir en un puerto de entrada/salida, se debe poder soportar tanto escribir en el puerto de lectura, como leer del puerto de escritura.
- Implementar en `Verilog` el *datapath* y la unidad de control del sistema. Identificar todas las conexiones entre los módulos. Tener en cuenta el criterio de nombres utilizado, tanto para las señales internas como externas de cada módulo. Además ordenar las señales de forma que sea simple identificar un error en el funcionamiento.
- Ejecutar los programas realizados en el ejercicio anterior y probar su correcto funcionamiento.

El resultado de este ejercicio debe ser un conjunto de archivos `Verilog` con el código implementado, archivos con los programas `asm` desarrollados y ejecutados, y un informe explicando toda la implementación del procesador.

2.4. Ejercicio 3: Organización en un solo ciclo

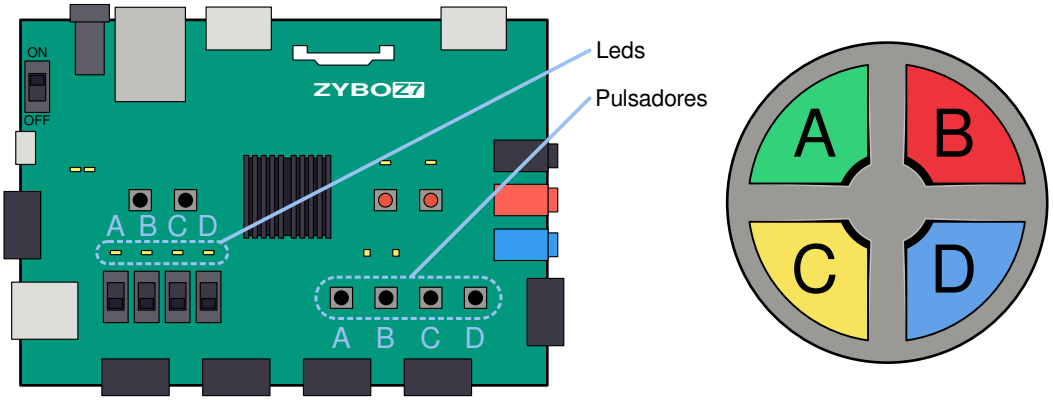


- Diseño del *datapath*, identificar todos los flujos de instrucciones y las dependencias entre cada uno de los componentes. Identificar que elementos del circuito cambian en el ciclo de reloj.
- Diseñar el flujo para incrementar y modificar el PC.
- Diseñar el flujo de decodificación de instrucciones, identificando que señales deben ir a la unidad de control y cuales deben ir al banco de registros.
- Diseñar el banco de registros de forma de soportar la cantidad de lecturas y escrituras simultaneas que hagan falta.
- Diseñar el flujo de datos desde la ALU u otras partes del *datapath* a el lugar donde se escriba el resultado de una operación, ya sea memoria, entrada salida o el mismo banco de registros.
- Diseñar el flujo de datos de la unidad de control y como las diferentes señales deben llegar a cada uno de los módulos del *datapath*. Agregar los multiplexores y decodificadores necesarios para resolver los flujos de datos y control.
- Implementar en `Verilog` cada uno de los componentes diseñados y su *datapath*. En todos los casos realizar *testbech* de las diferentes partes.
- Ejecutar los programas realizados en el ejercicio anterior y probar su correcto funcionamiento.

El resultado de este ejercicio debe ser un conjunto de archivos `Verilog` con el código implementado, archivos con los programas `asm` desarrollados y ejecutados, y un informe explicando toda la implementación del procesador.

2.5. Ejercicio 4: El juego

Desarrollar el código ensamblador del juego electrónico **Simon**. Las 4 salidas del puerto de salida serán conectadas a leds, mientras que las 4 entradas del puerto de entrada serán conectadas a pulsadores. Un pulsador y un led será utilizado para cada color del juego. Para reiniciar el juego se deberá presionar al menos dos pulsadores simultáneamente.



- Desarrollar el código ensamblador para la arquitectura implementada. Este código debe funcionar de forma equivalente en las dos organizaciones de procesador implementadas.
- Desarrollar el código `Verilog` para conectar los puertos de entrada/salida del procesador a los leds y pulsadores.
- Desarrollar el código de configuración de *constrains* para conectar el circuito a los puertos correspondientes de la placa de pruebas.
- Probar experimentalmente el código desarrollado en las dos organizaciones de procesador implementadas. Tener en cuenta que la temporización de los eventos en ambas organizaciones será diferente, por lo que es fundamental dejar el código ensamblador preparado para alterar las constantes de temporización en el caso de ser necesario.

El resultado de este ejercicio consistirá en probar su implementación en la placa FPGA. Se espera que puedan jugar al juego y recordar al menos 5 colores.

2.6. Ejercicio 5: Extensión de la Arquitectura

En este ejercicio se propone extender la arquitectura con al menos una de las siguientes características:

A. **Soporte para interrupciones**
Al menos se debe poder soportar una señal de interrupción que ejecute una sola rutina de código en una posición fija de memoria. La interrupción se debe generar ante el cambio de algún bit en un registro específico de entrada/salida.

B. **Soporte para protección de memoria**
Al menos el procesador debe soportar dos áreas de memoria distinguidas entre supervisor y usuario. El acceso a el área del supervisor estará limitado a código que ejecute con modo supervisor. Salir y entrar en modo supervisor solo se podrá hacer mediante un llamado específico a una rutina del sistema. La protección será válida tanto para lecturas y escrituras de datos como para ejecución de código.

C. **Soporte para pila**
Se debe dar soporte para las instrucciones de `push` y `pop` usando una pila implementada en memoria. Además estas instrucciones debe soportar tres modos de direccionamiento: inmediato, directo a registro y directo a memoria.

El resultado de este ejercicio consistirá en la documentación de la especificación del soporte propuesto. Se debe adjuntar en la documentación detalles de como se implementaría en las dos organizaciones presentadas en este trabajo práctico.

3. Entrega

Este trabajo práctico tendrá cuatro entregas o presentaciones. Las fechas de las entregas serán fijadas en el transcurso de la materia. En todos los casos se solicitará entregar un archivo pdf con el informe de la entrega correspondiente, con toda la información y código incluido. Adicionalmente los archivos de código deben ser adjuntados en otro archivo comprimido, junto con toda la información o documentación que se requiera para entender su solución particular.

- Entrega 1: Diseño de arquitectura
  Documentación de la arquitectura, definiciones de tamaños, parámetros. Descripción completa del conjunto de instrucciones con sus respectivos modos de direccionamiento. Detalle de la codificación de todas las instrucciones. Programa ensamblador y ejemplos de su funcionamiento.

- Entrega 2: Organización microprogramada
  Documentación del *datapath* completo. Descripción de su funcionamiento, ciclo de instrucción y flujo de ejecución de instrucciones. Código *Verilog* completo con su *testbech*.

- Entrega 3: Implementación en un solo ciclo
  Documentación del *datapath* completo. Descripción de su funcionamiento, ciclo de instrucción y flujo de ejecución de instrucciones. Código *Verilog* completo con su *testbech*.

- Entrega 4: El juego
  Implementación del juego en ensamblador. Pruebas de su funcionamiento en las dos organizaciones. Código `Verilog` para probarlo.

- Entrega 5: Extensión de la Arquitectura
  Esta entrega se presentará el último día de clases como parte de la evaluación final de la materia.