Código	FPI-009
Objeto	Guía de elaboración de Informe final de proyecto
Usuario	Director de proyecto de investigación
Autor	Secretaría de Ciencia y Tecnología de la UNLaM
Versión	5
Vigencia	03/9/2019

Departamento:

Ingeniería e Investigaciones Tecnológicas

Programa de acreditación:

CyTMA2

Código del Proyecto:

C2-ING-082

Título del proyecto

Implementación de la expansión de un procesador RISC-Vp en un entorno de desarrollo de lógica programable

Director:

Lic. Maidana, Carlos Eduardo

Integrantes:

Ing. Gho, Edgardo Alberto

Ing. Hnatiuk, Jair Ezequiel

Ing. Rodríguez, Carlos Alberto

Lic. Fiter, Jorge Antonio

Resolución Rectoral de acreditación: N°441/21

Fecha de inicio: 01/01/2021

Fecha de finalización: 31/12/2022

A. Desarrollo del proyecto (adjuntar el protocolo)

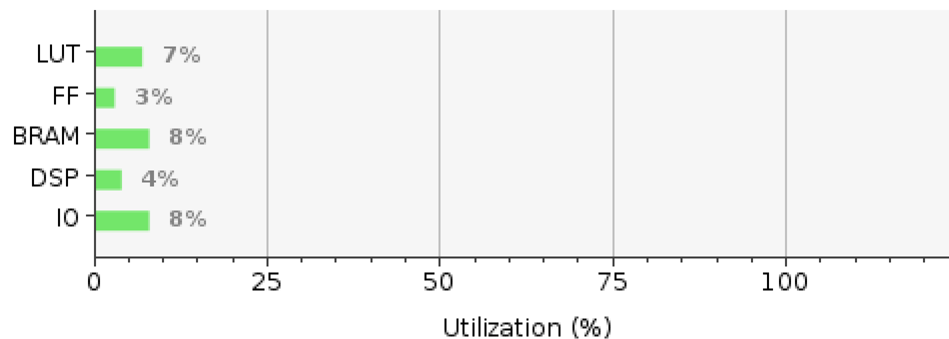
A.1. Grado de ejecución de los objetivos inicialmente planteados, modificaciones o ampliaciones u obstáculos encontrados para su realización (desarrolle en no más de dos (2) páginas)

Análisis de tareas realizadas y grado de cumplimiento

Se cumplió con todos los ítems planteados en el protocolo.

- Tomando como base el conocimiento obtenido al diseñar RISC-Vp (VHDL) se pudo re-implementar el mismo como RISC-V2p (Verilog). Esta nueva versión tiene significativas mejoras:
  - Todas las instrucciones se resuelven en un tiempo fijo de 4 ciclos. Esto deja abierta la posibilidad de implementar técnicas de pipelining sobre el mismo de ser necesario.
  - El diseño no solo puede ser simulado con herramientas de lógica programable, como era el caso de RISC-Vp (VHDL), sino que puede ejecutarse sobre silicio. Los ensayos se realizaron sobre el kit de desarrollo Digilent ARTY A7-35. El uso de recursos sobre este chip (Artix A7-35) es:

Resource	Utilization	Available	Utilization %
LUT	1368	20800	6.58
FF	1132	41600	2.72
BRAM	4	50	8.00
DSP	4	90	4.44
IO	16	210	7.62



El grado de utilización de recursos es menor que RISC-Vp (VHDL) dado que el diseño de RISC-V2p (Verilog) es enteramente sincrónico y la lógica de ejecución de instrucciones se implementó de forma “cableada” y no microprogramada evitando

implementar máquinas de estado finito las cuales consumen gran cantidad de recursos en su implementación.

Vale destacar que ciertos elementos de RISC-V2p (Verilog) pueden optimizarse aún más reduciendo el número de LUTs, en particular el módulo de generación de valores inmediatos que fue implementado con una cadena de multiplexores pero es posible mejorar esta implementación. Sin ir más lejos la implementación del mismo módulo en RISC-Vp (VHDL) es más eficiente pero más difícil de comprender, mientras que RISC-V2p (Verilog) consume más recursos en LUTs pero su implementación es de más sencilla comprensión, sobre todo teniendo en cuenta los fines didácticos de este trabajo.

Comparando RISC-Vp y RISC-V2p, podemos decir que:

	RISC-Vp (VHDL)	RISC-V2p (Verilog)
LUT	7,28%	6,58%
FlipFlops	2,84%	2,72%
BRAM	4%	4%
DSP	4,44%	4,44%

Claramente el número de BRAMs y DSP es idéntico ya que ambos implementan la misma cantidad de memoria y sumadores físicos. La implementación de RISC-V2p redujo el número de LUTs y FlipFlops necesarios, y si bien la diferencia parece escasa hay que considerar que RISC-V2p (Verilog) implementa interfaces de entrada/salida que RISC-Vp (VHDL) no posee, como la UART y el módulo SPI. Queda claro que el diseño de RISC-V2p (Verilog) es más eficiente que su antecesor.

- Uno de los problemas originales al desarrollar RISC-Vp (VHDL) fue que las herramientas de simulación no permiten hacer ciertos ensayos con tiempos post-implementación ya que solo están disponibles en diseños realizados en Verilog. Dado que RISC-V2p (Verilog) se encuentra enteramente realizado en Verilog fue posible realizar estos ensayos durante el desarrollo lo que facilitó la implementación sobre silicio. Podemos aclarar que sin estos ensayos hubiese sido imposible lograr que el diseño ejecute sobre la FPGA.

- El diseño de RISC-V2p (Verilog) fue ejecutado utilizando el oscilador incluido en el KIT Digilent ARTY A7-35 de 100MHz. El mismo pudo funcionar sin ningún inconveniente a esta

frecuencia. El camino con mayor demora es de 9ns. Consideramos entonces que la velocidad máxima teórica del diseño es de 111MHz. Estos valores son certeros ya que efectivamente se pudo implementar el diseño y no solo simularlo. En comparación, los valores simulados de RISC-Vp (VHDL) son de 321MHz, pero la naturaleza de este diseño de tipo simulado hace imposible obtener el equivalente en mayor demora post implementación, por ende la comparación es inválida. Es importante destacar que RISC-V2p (Verilog) está provisto de módulos de entrada/salida que su antecesor no posee, lo cual puede afectar también el máximo retardo de tiempo. En resumen, creemos que es más valioso disponer de un diseño que puede ejecutarse en silicio a 111MHz verificados, que un diseño simulado con un máximo teórico superior (pero no verificable). También suponemos que realizando las optimizaciones descritas en el punto anterior podemos lograr una mayor frecuencia máxima. Si se considera que un procesador Espressif ESP32-C3 (basado en RISC-V) posee una frecuencia máxima de operación de 160MHz, y el mismo es un ASIC que no tiene las limitaciones implícitas de un FPGA en cuanto a las demoras en el ruteo de lógica, obtener una frecuencia máxima de 111Mhz (casi 70%) es destacable.

- Se diseñó e implementó un controlador de memoria QSPI (trabajando en modo single). El mismo se verificó leyendo datos de la memoria QSPI de 16MB incluida en el kit de desarrollo Digilent ARTY A7-53. Esta misma memoria se utiliza como ROM de configuración del chip FPGA, por ende se reutilizan los pines de conexión, con la excepción del pin de clock que debe ser controlado en paralelo al módulo de configuración. Para la verificación del módulo SPI se escribió un programa en ASM de RISC-V que accede a todos los bytes de memoria mediante SPI y envía cada uno mediante el puerto serie UART a una PC. Luego se utilizó un lector de memorias SPI para leer el contenido de la memoria flash y se comparó byte a byte con el resultado obtenido del programa RISC-V. Ambos contenidos fueron idénticos. La frecuencia de reloj de la interfaz SPI se configuró a 1,25MHz pero este valor puede cambiarse si se dispone de memorias más veloces.

- Se implementó una memoria BRAM como memoria RAM principal. Si bien el planteo de la investigación buscaba implementar la misma en la memoria DDR3, se optó por integrar una BRAM Dual Port de forma tal que programas o datos almacenados en la memoria flash QSPI puedan ser ejecutados o accedidos por el procesador RISC-V2p (Verilog). Dado que la memoria DDR3 no posee la capacidad de ser Dual Port, se utilizó una del tipo BRAM. La capacidad de BRAM disponible es de 46.

- Se incorporó en el diseño una unidad UART totalmente sintetizable en Verilog, cuyo baudrate es de 115200 baudios con formato 8N1. Esta unidad es full-duplex permitiendo enviar y recibir al mismo tiempo. Existen bits de control y estado para indicar cuándo se

recibe información y cuando se termina de enviar la misma. Diversos programas fueron escritos en ASM de RISC-V para probar esta capacidad, incluyendo el arriba mencionado verificador de memoria flash que utiliza el módulo UART para hacer el barrido completo de la memoria.

- Durante el primer año de investigación se produjo un curso para el aprendizaje de técnicas de diseño digital sobre herramientas de lógica programable basadas en el lenguaje Verilog. Este curso fue utilizado como base de la materia Programación de Hardware de Ingeniería en Electrónica durante dos cuatrimestres consecutivos en 2022. El material gráfico de soporte se encuentra publicado en el anexo III de este informe. La experiencia de dictar el curso en ambas oportunidades fue satisfactoria, logrando que los alumnos puedan completar diseños digitales sincrónicos complejos en circuitos de alta frecuencia los cuales no pueden manejarse con las arquitecturas estudiadas en otras materias de la carrera como técnicas digitales 2 y 3. El conocimiento de herramientas de lógica programable y circuitos lógicos programables permite a los alumnos expandir sus capacidades en áreas donde las arquitecturas más pequeñas dejan un bache.

Una de las ventajas que tiene el procesador RISC-Vp (VHDL), desarrollado previamente por el grupo de investigación, es el soporte de accesos no alineados a memoria tanto para lectura como para escritura. Durante la etapa de diseño de RISC-V2p se decidió estudiar los beneficios de mantener este soporte. Se realizó una investigación sobre distintos chips con núcleo RISC-V a los que el grupo de investigación pudo tener acceso, descubriendo que solo algunas de las microarquitecturas RISC-V tienen soporte para esto. Estudiando profundamente qué alternativas existen para microarquitecturas que no dispongan de esta facilidad se desarrolló un algoritmo que permite resolver los accesos (con penalidad de tiempo) en aquellos casos donde se requiera hacer accesos no alineados pero la microarquitectura no lo soporte. Fruto de esto se publicó un trabajo en la revista Reddi, que se adjunta en el anexo I donde se detallan las arquitecturas estudiadas y el algoritmo de ejemplo. Es por esto que se decidió no incorporar soporte en la microarquitectura de RISC-V2p (Verilog) para este tipo de accesos ya que de ser necesarios pueden resolverse con el algoritmo planteado.

Producto también del trabajo realizado en el curso de lenguaje verilog, se presentó un trabajo en el workshop de investigadores en ciencias de la computación 2023 (WICC 2023), que se adjunta en el anexo I.

Todo el código Verilog de este proyecto se encuentra publicado en:

https://github.com/edgardogho/RISC-V2p.

El material gráfico del curso Verilog se encuentra publicado en :

https://github.com/edgardogho/CursoVerilog.

B. Principales resultados de la investigación

B.1. Publicaciones en revistas (informar cada producción por separado)

Artículo 1:	
Autores	*Gho, Edgardo Alberto*
Título del artículo	*Penalidades en lecturas no alineadas dentro de microcontroladores RISC-V*
N° de fascículo	*1*
N° de Volumen	*7*
Revista	*REDDI*
Año	*2022*
Institución editora de la revista	*Universidad Nacional de La Matanza*
País de procedencia de institución editora	*Argentina*
Arbitraje	SI
ISSN:	*2525-1333*
URL de descarga del artículo	*https://reddi.unlam.edu.ar/index.php/ReDDi/article/view/177/333*
N° DOI	*https://doi.org/10.54789/reddi.7.1.2*

B.2. Libros

Libro 1	
Autores	
Título del Libro	
Año	
Editorial	
Lugar de impresión	
Arbitraje	Elija un elemento.
ISBN:	
URL de descarga del libro	
N° DOI	

B.3. Capítulos de libros

Autores	
Título del Capitulo	
Título del Libro	
Año	
Editores del libro/Compiladores	
Lugar de impresión	
Arbitraje	Elija un elemento.
ISBN:	
URL de descarga del capítulo	
N° DOI	

B.4. Trabajos presentados a congresos y/o seminarios

Autores	*Edgardo Alberto Gho, Carlos Eduardo Maidana, Jair Ezequiel Hnatiuk.*
Título	*Adopción de Verilog en materias de diseño digital*
Año	*2023*
Evento	*WICC 2023*
Lugar de realización	*Universidad Nacional Noroeste Buenos Aires -UNNOBA*
Fecha de presentación de la ponencia	*13 y 14 de abril de 2023*
Entidad que organiza	*Universidad Nacional Noroeste Buenos Aires -UNNOBA*
URL de descarga del trabajo(especificar solo si es la descarga del trabajo; formatos pdf, e-pub, etc.)	*https://wicc2023.unnoba.edu.ar/2023/04/12/adopcion-de-verilog-en-materias-de-diseno-digital/*

B.5. Otras publicaciones

Autores	
Año	
Título	
Medio de Publicación	

C. Otros resultados. Indicar aquellos resultados pasibles de ser protegidos a través de instrumentos de propiedad intelectual, como patentes, derechos de autor, derechos de

obtentor, etc. y desarrollos que no pueden ser protegidos por instrumentos de propiedad intelectual, como las tecnologías organizacionales y otros. Complete un cuadro por cada uno de estos dos tipos de productos.

C.1. Títulos de propiedad intelectual. Indicar: Tipo (marcas, patentes, modelos y diseños, la transferencia tecnológica) de desarrollo o producto, Titular, Fecha de solicitud, Fecha de otorgamiento

Tipo	Titular	Fecha de Solicitud	Fecha de Emisión

C.2. Otros desarrollos no pasibles de ser protegidos por títulos de propiedad intelectual. Indicar: Producto y Descripción.

Producto	Descripción

D. Formación de recursos humanos. Trabajos finales de graduación, tesis de grado y posgrado. Completar un cuadro por cada uno de los trabajos generados en el marco del proyecto.

D.1. Tesis de grado

Director (apellido y nombre)	Autor (apellido y nombre)	Institución	Calificación	Fecha /En curso	Título de la tesis

D.2 Trabajo Final de Especialización

Director (apellido y nombre)	Autor (apellido y nombre)	Institución	Calificación	Fecha /En curso	Título del Trabajo Final

D.2. Tesis de posgrado: Maestría

Director (apellido y nombre)	Tesista (apellido y nombre)	Institución	Calificación	Fecha /En curso	Título de la tesis

D.3. Tesis de posgrado: Doctorado

Director (apellido y nombre)	Tesista (apellido y nombre)	Institución	Calificación	Fecha /En curso	Título de la tesis

D.4. Trabajos de Posdoctorado

Director (apellido y nombre)	Posdoctorando (apellido y nombre)	Institución	Calificación	Fecha /En curso	Título del trabajo	Publicación

E. Otros recursos humanos en formación: estudiantes/ investigadores (grado/posgrado/ posdoctorado)

Apellido y nombre del Recurso Humano	Tipo	Institución	Período (desde/hasta)	Actividad asignada[1]

F. Vinculación[2]: Indicar conformación de redes, intercambio científico, etc. con otros grupos de investigación; con el ámbito productivo o con entidades públicas. Desarrolle en no más de dos (2) páginas.

G. Otra información. Incluir toda otra información que se considere pertinente.

1Descripción de la/s actividad/es a cargo (máximo 30 palabras)

2Entendemos por acciones de "vinculación" aquellas que tienen por objetivo dar respuesta a problemas, generando la creación de productos o servicios innovadores y confeccionados "a medida" de sus contrapartes.

H. Cuerpo de anexos:

- Anexo I: Copia de cada uno de los trabajos mencionados en los puntos B, C y D, y certificaciones cuando corresponda.[3]
- Anexo II:
  - FPI-013: Evaluación de alumnos integrantes. (si corresponde)
  - FPI-014: Comprobante de liquidación y rendición de viáticos. (si corresponde)
  - FPI-015: Rendición de gastos del proyecto de investigación acompañado de las hojas foliadas con los comprobantes de gastos.
  - FPI-035: Formulario de reasignación de fondos en Presupuesto.
- Nota justificando baja de integrantes del equipo de investigación.

____

Lic. Carlos Eduardo Maidana
Director del Proyecto

Lugar y fecha: San Justo 17 de abril de 2023

- Cargar este formulario junto con los documentos correspondientes **exclusivamente** al Anexo I en SIGEVA UNLaM. Realizar la presentación impresa de los mismos junto con los restantes Anexos en la Secretaría de Investigación de la unidad académica correspondiente**.**

3En caso de libros, podrá presentarse una fotocopia de la primera hoja significativa o su equivalente y el índice.

ANEXO I

Revista Digital del Departamento de Ingeniería e Investigaciones Tecnológicas de la Universidad Nacional de la Matanza
ISSN: 2525-1333
Vol.: 7 - Nro. 1 (AGOSTO 2022)

Artículo original

Penalidades en lecturas no alineadas dentro de Microcontroladores RISC-V

Misaligned memory loads penalties in RISC-V Microcontrollers

Edgardo Gho

Universidad Nacional de La Matanza
Grupo de investigación en lógica programable
egho@unlam.edu.ar

Universidad Abierta Interamericana
EdgardoAlberto.Gho@alumnos.uai.edu.ar

Resumen:
La arquitectura RISC-V fue concebida con el fin de evitar los problemas de sobrecarga de instrucciones de las arquitecturas x86 y ARM. Su definición es abierta dejando librado los detalles de la microarquitectura al diseñador del procesador. Las implementaciones de microcontroladores RISC-V se comportan de manera distinta en cuanto a los accesos a datos de forma no alineada. Si bien los compiladores buscan evitar este tipo de accesos, determinadas estructuras de datos requieren los mismos en ámbitos donde la memoria es limitada. En este artículo se estudia la implementación de tres microarquitecturas RISC-V en cuanto a los accesos a

memoria no alineados y se plantea un código que permite salvar la ejecución de programas que realizan accesos no alineados cuando la microarquitectura no tiene soporte para los mismos. En los casos donde la microarquitectura soporta accesos no alineados se estudia el impacto en la eficiencia de ejecución de instrucciones.

Abstract:
The RISC-V instruction set architecture is designed to overcome current problems of bloated instruction sets present in other architectures like x86 and ARM. The ISA does not restrict micro architecture implementations leaving those details free to the chip designer. RISC-V microcontrollers can then provide support for misaligned memory loads or depend on software emulation. Compilers will try to prevent these types of memory accesses nevertheless some data structures require them mostly on memory constraint systems. This article studies three different RISC-V microarchitecture implementations related to misaligned accesses and provides code to perform software emulation when the microarchitecture does not support them. Instruction execution penalties are studied comparing them to aligned memory accesses.

Palabras Clave: Arquitectura de computadoras, RISC-V, estructura de datos
Key Words: Computer architecture, RISC-V, data structures
Colaboradores: Carlos Maidana, Jair Hnatiuk

ISSN: 2525-1333. Vol.: 7 - Nro. 1 (AGOSTO-2022)

I. INTRODUCTION

A computer is composed of three main components, being the Central Processing Unit (CPU), Memory Unit and Input/Output(I/O). Memory content is defined by the user depending on the application. Programs designed to be executed by the CPU need to be stored in memory for the CPU to access and execute. This is done by storing the program instructions on a memory section, which the CPU accesses sequentially during the execution of the program. Program data, usually referred to as variables, are also stored in memory and accessed by the CPU in the same manner. Computer architecture defines a data unit named *word* which represents the size of a single unit of information. The unit size is usually tailored to the application. Most modern computer architectures [1] have *word* sizes larger than one byte, being 4 bytes (32 bits) and 8 bytes (64 bits) the more common. Some computer architectures, in particular Harvard-based ones, define different memory ranges to differentiate between program instructions and program data. These ranges need to be known in order to properly use the computer. This information is usually represented on a memory map which lists the ranges where a program instructions and data can be stored. The memory addressing is part of the computer architecture and virtually all computer architectures [2] use byte addressing. In essence, byte addressing allows the user to reference a single byte inside a *word*. Nevertheless, memories are organized as block arrays and are meant to be accessed one block at a time. The block size is related to cache memory line size [3], in particular L1. So even though CPUs support byte addressing, the program information (being instruction or data) is usually word aligned, so much so that some CPU architectures enforce a restriction preventing memory accesses that cross the *word* boundary. These types of accesses are usually referred to as unaligned or misaligned accesses.

In [3] the authors study the impact of different unaligned memory accesses on different x64 ISA (instruction set architecture) instructions. Depending on the boundary crossed the time penalty can be up to 1800%.

The RISC-V is an open standard instruction set architecture that is not bound to a particular implementation [4]. This means that while the computer architecture is standardized and clearly defined, manufacturers have no restrictions on how to implement the ISA. While this gives an enormous amount of freedom to the designer it can create fragmentation among different implementations of the same ISA [5]. Being an open architecture RISC-V allows microarchitecture changes to improve security while maintaining the base instruction set compatibility.

This paper studies the microarchitecture differences within memory access alignment in three different RISC-V CPUs. Although the three CPUs implement the same basic RV32I instruction set as a minimum, some implement a more complete RV32IMAC set including integer multiplication/division, atomic access and compressed instructions. The focus of the paper is comparing how the three handle misaligned load access to data memory. As a result, we propose a simple code to handle misaligned loads for *word* size data, both in RV32I and RV32IMAC sets. A benchmark is performed to measure the execution time penalty.

It is important to highlight that most compilers will try to prevent misaligned access to memory. One known case is

ISSN: 2525-1333. Vol.: 7 - Nro. 1 (AGOSTO-2022)

the GCC [6] compiler which adds padding to data structures in order to keep load access to the data structure element aligned. Even though this will prevent misaligned access, the padding added to the data structure consumes memory. On CPU implementations designed for embedded systems and not general purpose computers memory is usually constrained and using padding in data structures will waste this resource.

II. BACKGROUND

A. RISC-V

The base RISC-V ISA [7] is a fixed 32 bit instruction length. However, it is designed to be extended to a variable length set using 16 bits parcels. Therefore, instruction alignment needs to be naturally aligned to the 16 bit parcel size. If the implementation is restricted to the basic RV32I set, then a 32 bit natural alignment is needed for instructions, although it is very common to find RV32IC which supports compressed instructions changing the requirement to a 16 bit natural alignment. In fact, the macro-op fusion benefits [8] from using the compressed instructions to achieve better performance than other ISAs like x86 or ARM.

Data access on the other hand is byte addressed and there are no restrictions by the ISA in terms of alignment. The authors describe that for best performance data access should be naturally aligned with the data size. This means that words (loaded with LW) should be aligned to 32 bits and half words (loaded with LHW or LHU) should be aligned to 16 bits. Even though this is suggested for best performance, it is ultimately up to the chip designer to support misaligned access. The RISC-V ISA does not impose a restriction on the microarchitecture, so chip designers are free to choose to support it or not. Nevertheless, the ISA defines a trap mechanism to emulate the access in software. This means that it is ultimately possible to support these types of misaligned access but with a time penalty.

B. Compilers

A typical compiler like GCC is normally aware of the microarchitecture implementation allowing the programmer to remain ignorant about it. In the case of RISC-V, there is a set of registers accessible using the CSR* instructions which describe the microarchitecture. The *misa* register will describe which of the extensions the CPU supports. Therefore, it is possible for a compiler to prepare code for several microarchitectures and select the correct one during runtime. Unfortunately, this will create bloated binaries [9] which are not memory efficient on memory constrained devices like microcontrollers. A clear example of this is the M extension. This extension provides hardware implementation for multiply and divide instructions which should be faster than a software implementation. Hence the compiler can prepare the code to detect if the M extension is supported during runtime and implement a software emulation alternative in case it does not. To facilitate this process the RISC-V architecture provides traps for invalid instructions. This would allow the compiler to provide a trap handler that would handle the missing instructions on the basic RV32I set. There is of course a penalty for this but other than the trap handler the original code remains the same.

The same technique can be used to handle other faults related to the microarchitecture. In this paper a proposed software emulation for a misaligned load is provided.

Revista Digital del Departamento de
Ingeniería e Investigaciones
Tecnológicas de la Universidad
Nacional de la Matanza

ISSN: 2525-1333. Vol.: 7 - Nro. 1 (AGOSTO-2022)

```
struct Data {
    uint8_t   A;
    uint32_t B;
    int16_t   C;
    int8_t     D;
    uint8_t   E;
};
```

Fig. 1 Misaligned Data Structure

C. Data Structures

It is normal for programmers to group related information using data structures. The C programming language struct is a good example of this. Looking at Figure 1, the data structure is composed of an 8 bit unsigned integer named A, a 32 bit unsigned integer named B, a 16 bit signed integer named C, an 8 bit signed integer named D and finally an 8 bit unsigned integer named E.

If the programmer instantiates an element with this data structure the compiler will be aware that the access to certain elements will be misaligned in 32 bits systems [10]. If the structure is stored on a byte addressing computer in a way that naturally aligns the first element (A), it will produce a misaligned access to the B and C elements.

A programmer that is aware of the microarchitecture limitations has the option to arrange the data in a way to force natural alignment for the elements that are bigger than 8 bits. So, Figure 2 would be a possible re-ordering of the original structure in a way that all access to the structure elements are naturally aligned.

```
struct Data {
    uint32_t B;
    int16_t   C;
    uint8_t   A;
    int8_t     D;
    uint8_t   E;
};
```

Fig. 2 Aligned Data Structure

Unfortunately requiring this change would be cumbersome to the programmer since it needs to manually align the elements of the data structure, but it's still not good enough. The re-arrangement will work for a single instance of the structure assuming the instance is stored in a naturally aligned address but since the number of bytes composing the structure is odd, it will not allow saving two consecutive instances like an array. The second instance will be misaligned.

For this reason, compilers will not require programmers to naturally align elements inside a structure and will provide a solution to the alignment access by adding padding to the structure. On Figure 1, the compiler will reserve three consecutive bytes after element A effectively expanding A to be 32 bits. That way the structure is now naturally aligned for all elements access but now it uses 12 bytes instead of the original 9 bytes. This is a +33% increase penalty to keep alignment. Furthermore, if the structure would be composed only by element A and B, the added padding would increase the original structure to a +60%.

Compilers provide ways to restrict padding by forcing the data to be packed, which does not add any padding, but it does not enforce alignment. It is up to the programmer to choose whether to save memory space or improve speed by avoiding misaligned access which might incur in time penalties.

III. RELATED WORK

The Linux kernel recently added support for software emulated misaligned accesses [11], but this implementation requires running the Linux kernel which is not always an option specially on systems that have no

ISSN: 2525-1333. Vol.: 7 - Nro. 1 (AGOSTO-2022)

Table 1 RISC-V Microcontrollers

CPU	FE310-G002	ESP32C3	RISC-Vp
Execution	In-order pipelined	In-order pipelined	In-order no pipelined
Extensions	RV32IMAC	RV32IMC	RV32I
Memory	16KB data. 16KB instruction. Instruction memory expandable using external Flash.	400 KB shared for data and instructions. Instruction memory expandable using external Flash.	Up to 540KB using FPGA BlockRAM shared data and instruction. No expansion.
Misalignment	Software emulation	Hardware support	Hardware support

MMU and have fixed limited memory. Microcontrollers like the ones studied on this paper usually implement software in bare-metal or using a RTOS. The provided code in this paper can be easily adapted to these chips without imposing a full Linux kernel implementation.

The authors in [12] study the effects of misaligned memory access in microcontrollers but it is not restricted to RISC-V. Their study also includes ARM and MIPS architectures. Their study is restricted in some cases to byte access since not all platforms allow misaligned access for words. This paper focuses on word load penalty when address is misaligned and provides software emulation when hardware does not support this type of access.

Misaligned accesses become important based on [3] Heap randomization can be improved in architectures that support misaligned accesses.

IV. EVALUATION

Three different RISC-V microarchitectures were studied in terms of their respective misaligned loads. All three processors are oriented to microcontroller applications, therefore memory is usually fixed size and as a scarce resource it is important to maximize its usage. Hence using padding in data structures is not recommended. Table 1 represents the instruction set extensions, memory size and microarchitecture details of the three processors.

A. FE310-G002

This RISC-V microcontroller [13] only supports software emulated misaligned loads of words. The code present in Figure 3 shows how to force a misaligned load using pointers.

The C compiler translates the misaligned load of pWord pointer into loadVariable with the ASM listed on Figure 4.

The middle ASM instruction in the group is the actual misaligned load. Since it's a full 32 bit word load with offset 0 based on the address pointed by a5 into a5, this instruction can be compressed into 16 bits.

In order to test time penalties, a trap handler was written to software-emulate the misaligned access. The code for

ISSN: 2525-1333. Vol.: 7 - Nro. 1 (AGOSTO-2022)

```
struct Data {
   uint32_t A;
   uint32_t B;
};
struct Data data;
data.A=0x12345678;
data.B=0x90ABCDEF;
uint8_t *pByte = (uint8_t*)&data;
uint32_t *pWord = (uint32_t*)(pByte+4);
uint32_t loadVariable = *pWord; //Aligned
pWord = (uint32_t*)(pByte+1);
loadVariable = *pWord; //Misaligned
```

Fig. 3 Misaligned pointer access

the handler is provided in [14]. This handler only supports LW instructions, although it can be easily extended to handle LH and LHU misaligned access as well. The handler supports both RV32I and RV32IC extensions, detecting at runtime if the offending instruction is LW or C.LW (compressed LW).

The handler reads the *mcause* register to check if it is a load alignment issue. If that is the case, then it recovers the address of the offending instruction from the *mepc* register. Since the instruction can be either LW or C.LW, it can be aligned to 32 or 16 bits, so the handler supports the possible misaligned instruction access. Upon reading the instruction it calculates the return address for the trap (done later with *mret*) and the destination register for the load. In order to emulate the access, several registers are temporarily used, and their previous content is saved on the stack. Upon return the original content of those will be recovered, except for the destination register of the misaligned access which should have the misaligned data. This is true for all cases except when the destination register is the stack pointer itself. In this case, there needs to be an extra scratch register or fixed memory location

```
lw   a5, -24(s0)
lw   a5,   0(a5)
sw   a5, -28(s0)
```

Fig. 4 ASM translation

used to store the misaligned value prior to calling *mret*. In the case of this handler, it will scratch the T5 register, but this can be modified to use a fixed memory location and avoid using T5.

The software emulation executed 92 extra instructions for the access listed on Figure 4. This number of extra instructions will depend on how the original offending instruction is coded, what type of misalignment it performs and most important what is the destination register, being the higher ones from x0 to x31 the worst ones.

B. ESP32C3

This RISC-V microcontroller [15] supports hardware misaligned loads. There is no need to provide a trap handler since the hardware supports the misaligned access. Since ESP32C3 has a pipelined microarchitecture with data and instruction caches, using the cycle performance counter (CSR 0x7e2 for this chip) might yield different values depending on the pipeline stage and cache state. Therefore, a loop doing 100 aligned access followed by another loop with 100 misaligned access were executed while saving the cycle performance counter. The results yielded 807 cycles for aligned access and 1906 cycles for the misaligned ones. These numbers include the loop instructions and performance counter access overhead but since it is the same overhead on both loops the comparison is still valid. The misaligned access incurs a 137% penalty.

Table 2 Results

CPU	FE310-G002	ESP32C3	RISC-Vp
Penalty	9200%	137%	75%
CPI	1	1	4
C100LW	9300	237	700

C. RISC-Vp

This RISC-V implementation [16] also supports hardware misaligned loads. Since this microarchitecture was designed as an academic example of a RISC-V implementation it is possible to predict the penalties in misaligned access.

A proper aligned load uses 4 clock cycles while a misaligned load consumes 7 clock cycles. This is a 75% penalty when compared with an aligned load. This microarchitecture executes on a fixed clock per instruction since it has no pipeline and no cache, therefore there is no need to execute loops or use performance counters.

V. CONCLUSIONS

Table 2 shows that software emulation can result in a big time penalty versus the hardware implementation. The penalty for FE310-G002 is estimated since the trap handler efficiency varies depending on the offending load. Both LW and C.LW were tested. The CPI (clocks per instruction) represents the non-pipelined architecture of RISC-Vp with a fixed 4 clock per instruction. The CPI listed for FE310-G002 and ESP32C3 is also estimated since it would depend on the pipeline state depending on the code being executed but the goal for the pipeline would be one clock per instruction.

Even though RISC-Vp has a lower time penalty than ESP32C3 which also supports hardware misaligned loads, ultimately the number of clocks per 100 misaligned loads (C100LW) ends up being less for ESP32C3 due to the pipeline implementation.

The result clearly indicates that software emulation of misaligned load can incur severe time penalties affecting performance. Systems that require misaligned access should try to select a microarchitecture that can perform these without software emulation.

The provided sample code [14] can be expanded to support misaligned stores. This code is a good example of software emulation for missing microarchitecture features proving that RISC-V was designed in a way that simpler microarchitectures can execute code for more complete implementations. In this test scenario the FE310-G002 was limited in terms of misaligned access but this limitation can be overcome via software emulation at a high time penalty.

VI. REFERENCES

[1] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach Fifth Edition. 2007.

[2] D. M. Harris and S. L. Harris, Digital design and computer architecture, 2nd edition. 2012. doi: 10.1016/C2011-0-04377-6.

[3] D. Jang, J. Kim, M. Park, Y. Jung, H. Lee, and B. B. Kang, “Rethinking Misalignment to Raise the Bar for Heap Pointer Corruption.” arXiv, 2018. doi: 10.48550/ARXIV.1807.01023.

[4] K. Asanović and D. Patterson, “RISC-V: An Open Standard for SoCs | EE Times,” EE Times, 2014.

[5] Agam Shah, "RISC-V takes steps to minimize fragmentation" https://www.theregister.com/2022/04/01/riscv_fragmentation/, Apr. 01, 2022.

[6] R. M. Stallman and T. G. D. Community, "Using the GNU Compiler Collection," Development, vol. 2. 2012.

[7] A. Waterman et al., "The RISC-V instruction set manual," Volume I: User-Level ISA', version, vol. 2, 2014.

[8] C. Celio, P. Dabbelt, D. A. Patterson, and K. Asanović, "The Renewed Case for the Reduced Instruction Set Computer: Avoiding ISA Bloat with Macro-Op Fusion for RISC-V." arXiv, 2016. doi: 10.48550/ARXIV.1607.02318.

[9] A. Singh, Mac OS X Internals: A Systems Approach (paperback). Addison-Wesley Professional, 2006.

[10] Daniel Lemire, "Data alignment for speed: myth or reality?," https://lemire.me/blog/2012/05/31/data-alignment-for-speed-myth-or-reality/, May 31, 2012.

[11] Damien Le Moal, "[v2,1/9] riscv: Unaligned load/store handling for M_MODE" https://patchwork.kernel.org/project/linux-riscv/patch/20200312051107.1454880-2-damien.lemoal@wdc.com/, Mar. 12, 2020.

[12] M. Hubacz and B. Trybus, "Data Alignment on Embedded CPUs for Programmable Control Devices," Electronics (Basel), vol. 11, no. 14, 2022, doi: 10.3390/electronics11142174.

[13] SiFive Inc, "SiFive FE310-G002 Manual v1p4," 2019.

[14] Edgardo Gho, "RISC-V Traps," https://github.com/edgardogho/RISC-V-Traps, Jul. 18, 2022.

[15] Espressif Systems, "ESP32-C3 Series Datasheet," 2022.

[16] Edgardo Gho, "RiscVP," https://github.com/edgardogho/RiscVP, Mar. 04, 2021.

Recibido: 2022-07-20
Aprobado: 2022-08-02
Hipervínculo Permanente: https://doi.org/10.54789/reddi.7.1.2
Datos de edición: Vol. 7 - Nro. 1 - Art. 2
Fecha de edición: 2022-08-10

Adopción de Verilog en materias de diseño digital

Gho Edgardo, Maidana Carlos, Hnatiuk Jair
Universidad Nacional de la Matanza
Universidad Abierta Interamericana

GILP
GRUPO INVESTIGACION LOGICA PROGRAMABLE

1. Contexto

La presente investigación surge de los proyectos I/D del Departamento de Ingeniería e Investigaciones Tecnológicas de la UNLaM C2-ING-082 "Implementación de la expansión de un procesador RISC-Vp en un entorno de desarrollo de lógica programable" y de su proyecto antecesor del programa PROINCE C-219 "Desarrollo e implementación de una arquitectura basado en el conjunto de instrucciones RISC-V". Los proyectos fueron llevados a cabo por el Grupo de Investigación en Lógica Programable (GILP) y se vinculan con la materia de la carrera de Ingeniería en Electrónica "Programación de Hardware" y la materia "Arquitectura de Computadoras" de la carrera de Ingeniería en informática del mencionado departamento.

2. Línea de investigación

A lo largo del desarrollo del núcleo RISC-Vp escrito en VHDL surgieron problemas fruto de la utilización del software Vivado, ya que los modelos de los chips necesarios para hacer simulaciones post-síntesis y post-implementación no se encuentran disponibles para VHDL. Esto es una limitación de la biblioteca SIMPRIM que solo posee modelos de retardos de tiempos en Verilog. Dado que al diseñar una unidad central de procesos para correr dentro de un circuito lógico programable es imperativo verificar el cumplimiento de las restricciones de tiempos tal que todo funcione de forma sincrónica, no disponer de esta herramienta en VHDL generó serias complicaciones. Es por ello que se planteó un proyecto de investigación con el fin de migrar el diseño del núcleo RISC-Vp a Verilog. Dentro de este proyecto se logró:

- Conocer la sintaxis de Verilog
- Discernir las diferencias entre Verilog y VHDL
- Implementar circuitos lógicos utilizando Verilog en las herramientas de desarrollo de Xilinx.

Considerando que el grupo de investigación posee larga experiencia utilizando VHDL en aplicaciones prácticas pero también como herramienta en el dictado de clases y cursos en lógica programable, se procedió a diseñar un curso introductorio al lenguaje Verilog que pudiese ser dictado a alumnos de la carrera de Ingeniería en Electrónica. Se decidió utilizar Verilog en lugar de SystemVerilog ya que es el conjunto mínimo soportado por la mayoría de las herramientas. Es digno de mención que ciertos kits de desarrollo más antiguos solo funcionan con herramientas de programación descontinuadas, las cuales soportan Verilog pero no necesariamente SystemVerilog. Por cuanto los mencionados kits descontinuados se siguen empleando en ciertos ámbitos, debido principalmente por el alto costo de reemplazarlos, se consideró conveniente preparar un curso que soportara un lenguaje compatible con todos los kits.

Teniendo como base la experiencia de haber dictado cursos similares utilizando VHDL se propusieron los siguientes objetivos

- Introducir al alumno en la historia del diseño digital.
- Plantear la necesidad de los lenguajes descriptivos de hardware como alternativa superadora a la captura digital de esquemáticos.
- Describir el funcionamiento de los circuitos lógicos programables.
- Presentar de forma concreta y resumida los elementos mínimos para realizar un diseño elemental y de baja complejidad como introducción a la sintaxis y las herramientas de desarrollo en Verilog.
- Incrementar el nivel de conocimiento agregando herramientas tales como simulaciones y funciones de ayuda provistas por el lenguaje.
- Realizar prácticas de circuitos combinacionales.
- Introducir circuitos secuenciales y los distintos tipos de asignaciones provistas por el lenguaje.
- Describir los elementos más avanzados del lenguaje, como la instanciación de elementos físicos, secuencias encadenadas y parametrización de módulos.
- Introducir al diseño e implementación de módulos de propiedad intelectual (IP Cores).
- Estudiar circuitos complejos como máquinas de estado.

3. Resultados obtenidos

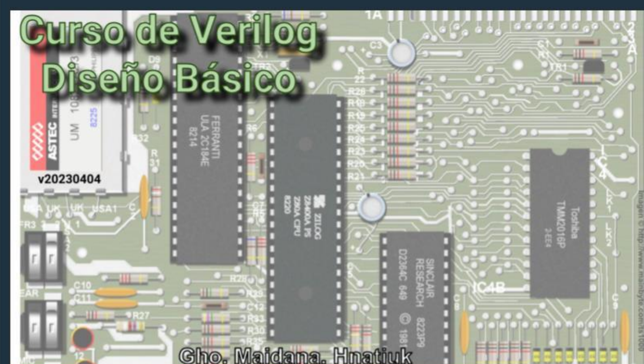
Los resultados del proyecto de investigación han sido satisfactorios. Se diseñó el curso y el mismo fue utilizado por primera vez durante el primer cuatrimestre de 2022 en la materia Programación de Hardware de la carrera Ingeniería en Electrónica. Sobre el mismo se hicieron ciertas mejoras, particularmente al abordar las máquinas de estado finito, que fueron utilizadas en el segundo cuatrimestre de 2022. El material de apoyo visual se encuentra publicado en https://github.com/edgardogho/CursoVerilog.

El curso se dicta en laboratorio con acceso al software de Xilinx Vivado, versión 2019 en adelante, y kits de desarrollo de la familia FPGA Xilinx series 7, como el Artix 7 o Spartan 7. A los alumnos se les exige realizar un proyecto integrador donde demuestren sus conocimientos del tema y utilicen algunos de los componentes más avanzados de los kits de desarrollo. El objetivo es cubrir la brecha que existe entre proyectos realizables con un microcontrolador de bajo porte versus proyectos involucrando un sistema de computación completo basado en microprocesadores con costos elevados donde se depende de un sistema operativo. Considerando que la aplicación más factible de circuitos lógicos programables es en aceleradores de procesos, se busca que los alumnos implementen interfaces a medida o algoritmos traducidos de software a hardware.

La meta para la siguiente etapa consiste en agregar nuevo material focalizándose en el diseño sincrónico, considerando las líneas de retardo de tiempos para mantener la sincronía dentro de los FPGA. Las ventajas de utilizar SystemVerilog quedan planteadas para una versión avanzada del curso en un futuro proyecto de investigación.

Dada la familiaridad del grupo de investigación tanto con VHDL como con Verilog, y el hecho de haber realizados similares diseños en ambos lenguajes así como haber dictado cursos en los mismos, creemos importante destacar que Verilog es un típico lenguaje simple de escribir, es decir, se logra escribir mucho código en un tiempo reducido ya que la sintaxis no es tan estructurada como VHDL. Sin embargo, a la hora de leer el código VHDL resulta más simple de comprender en una primera mirada, teniendo características de ser un lenguaje difícil de escribir pero sencillo de leer. Esta impresión está en línea con lo planteado sobre la inspiración de lenguajes de programación tradicionales como lo es el lenguaje C que se observa en Verilog , mientras que VHDL sigue una estructura en la línea de otros lenguajes fuertemente tipados como ADA.

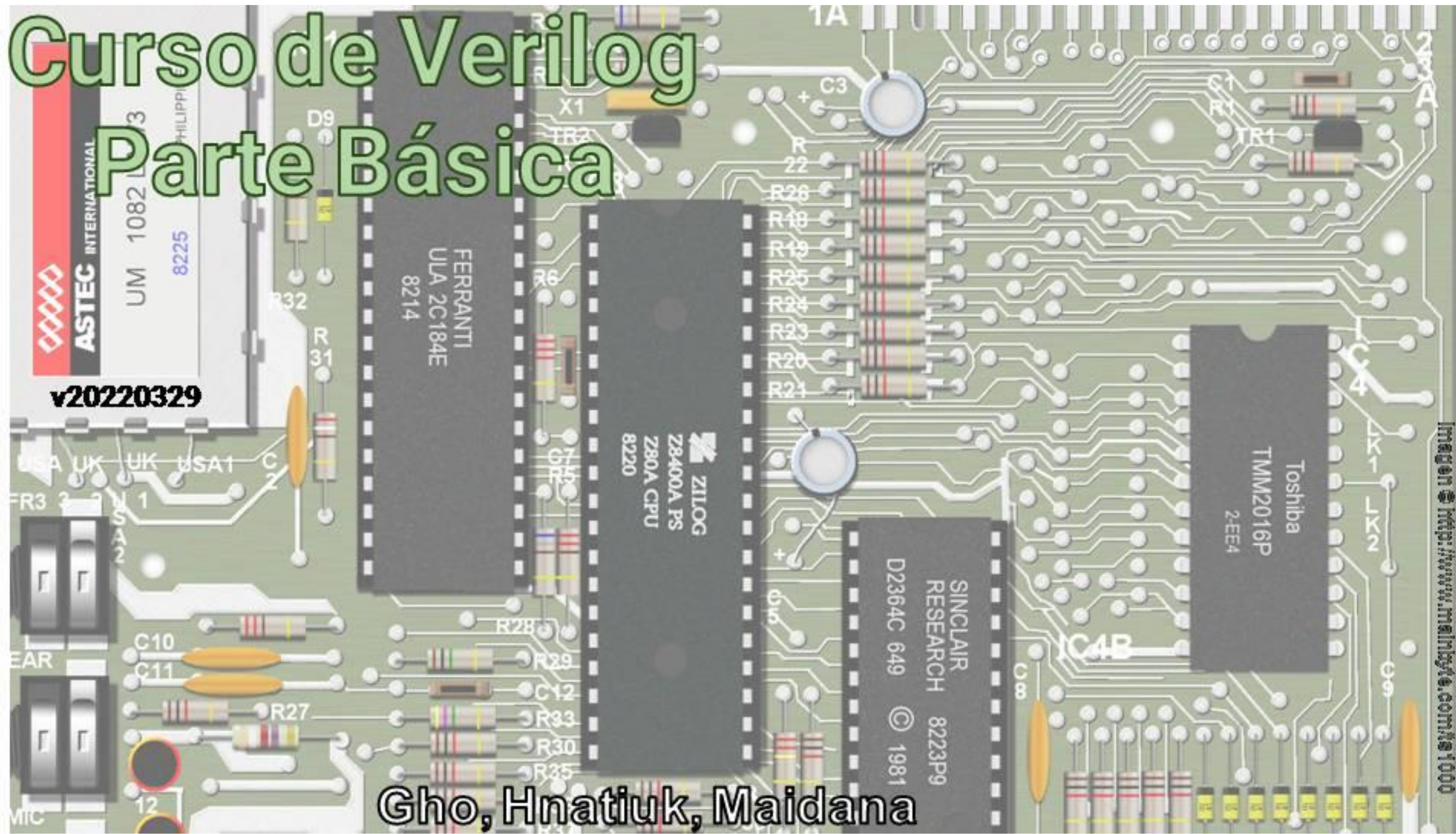
También es importante destacar que el proyecto que originó los resultados aquí descritos consiste en diseñar una microarquitectura basada en el set de instrucciones RISC-V utilizando Verilog. Este proyecto tuvo como resultado el procesador RISC-V2p publicado en https://github.com/edgardogho/RISC-V2p



4. Formación de recursos humanos

En el proyecto trabajan docentes investigadores del Departamento de Ingeniería e Investigaciones Tecnológicas de la Universidad Nacional de La Matanza. La mayoría son docentes en carreras tanto de Informática como de Electrónica. Uno de los miembros se encuentra realizando un Doctorado en Informática en la Universidad Abierta Interamericana sobre Arquitecturas de Computadora utilizando como herramienta de prototipado y pruebas Verilog para el modelado de arquitecturas basadas en el set de instrucciones RISC-V.

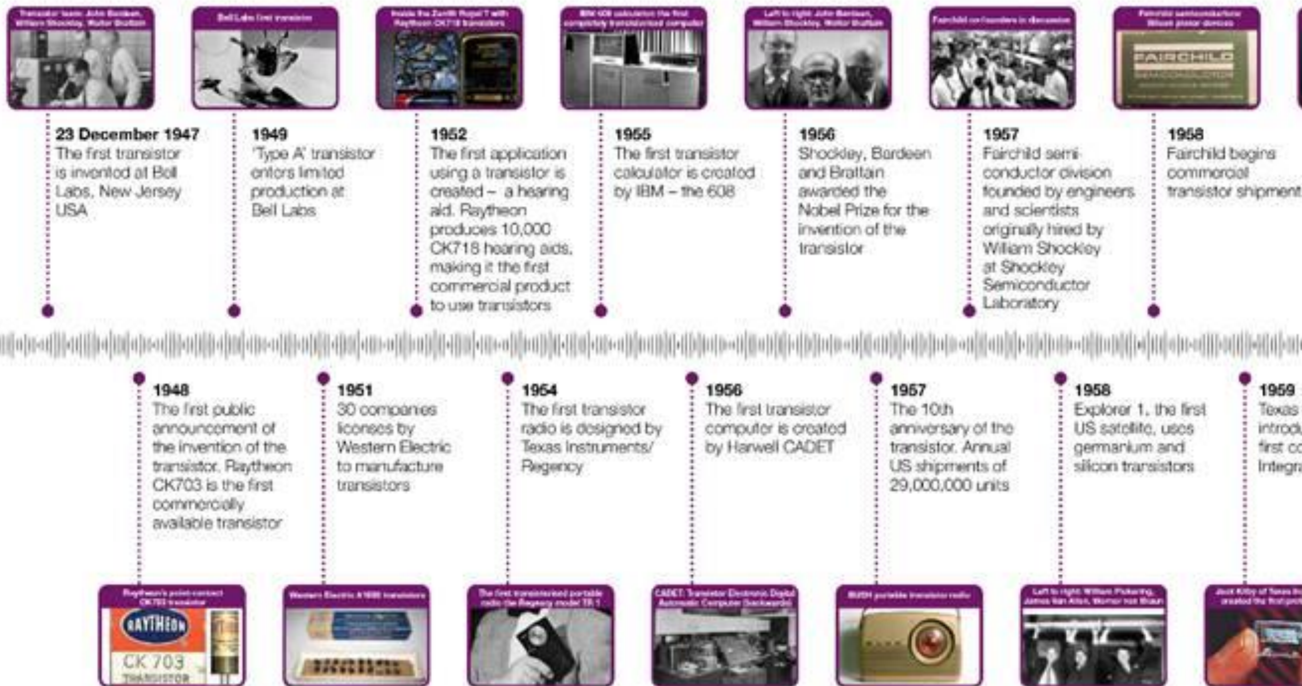
ANEXO III


Curso de Verilog
Parte Básica
v20220329
Gho, Hnatiuk, Maidana
imagen © https://www.mainbyte.com/ts1000

Diseño de circuitos

Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6

Celebrating 70 years of the transistor

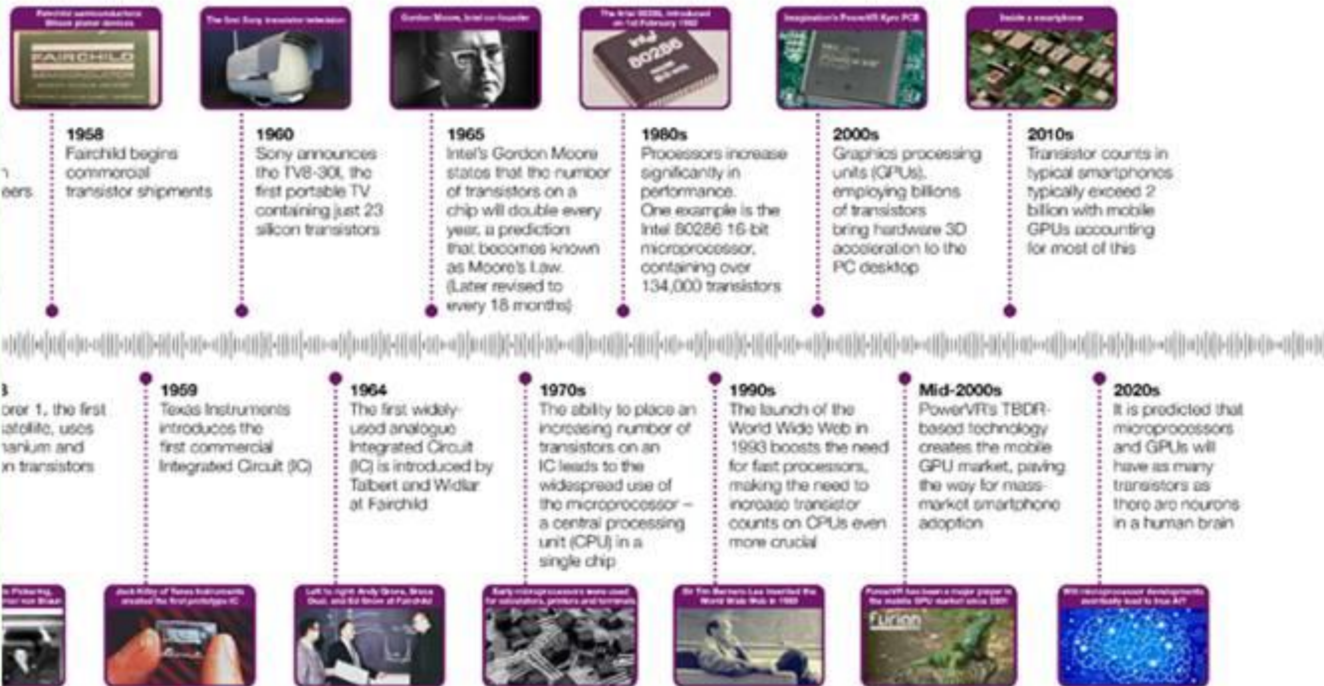


https://www.electronicsweekly.com/news/first-transistor-created-70-years-ago-the-device-that-changed-the-world-2017-12/

Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6
Curso de Verilog - Básico

La lógica programable

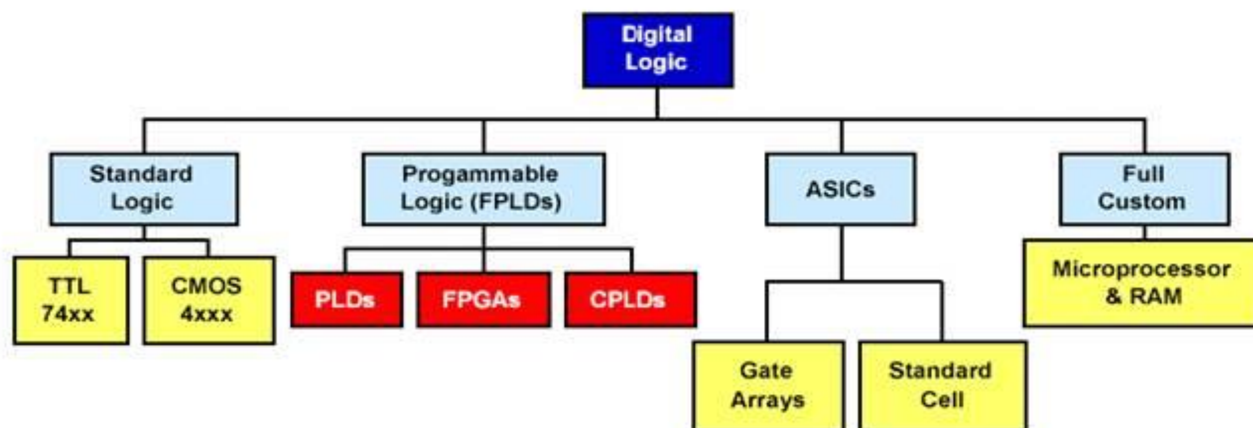
tor



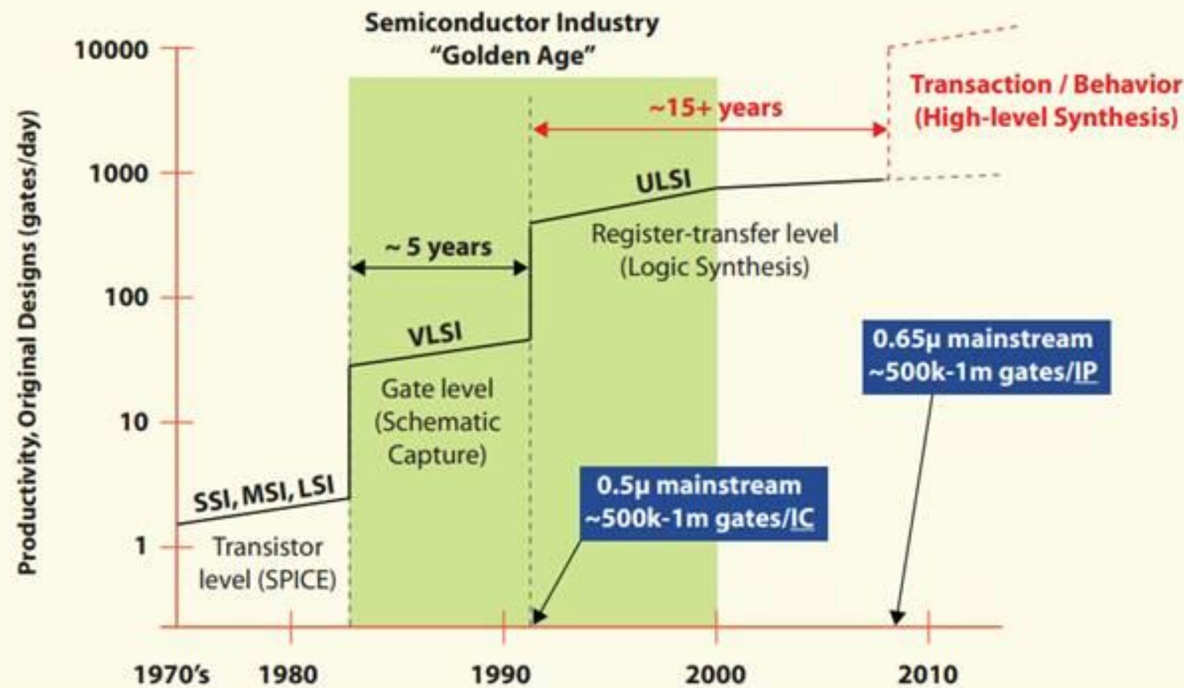
https://www.electronicsweekly.com/news/first-transistor-created-70-years-ago-the-device-that-changed-the-world-2017-12/

Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6

Diseño digital



La lógica programable



https://www.accellera.org/images/resources/articles/icdesigntrans/ic_design_transition_feb2010.pdf

HDL (Lenguajes descriptivos de hardware)

VHDL

Se desarrolla en 1983 como un lenguaje para describir el funcionamiento de circuitos. El problema era que en los 80s la documentación de cómo se "comportaban" los C.I. variaba mucho según el fabricante. El depto de defensa de USA necesitaba un estándar y entonces surge VHDL. En 1987 se estandariza como IEEE 1076-1987. A partir de este punto los fabricantes de circuitos y herramientas CAD empiezan a sugerir cambios y mejoras que terminan en IEEE 1076-1993.
A lo largo de los años se ha ido actualizando el estándar soportando cambios en el lenguaje.
Uno de los cambios principales que incorpora la versión de 1993 es la síntesis de circuitos utilizando el mismo lenguaje. Si bien existe sintaxis específica para simular y sintaxis específica para sintetizar, el lenguaje es prácticamente el mismo para ambas tareas.

Verilog (Verification-logic) IEEE 1364

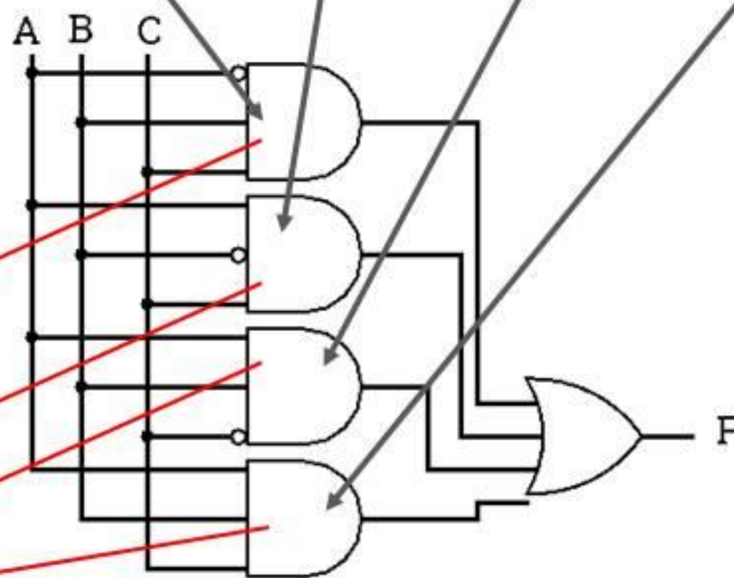
Creado en 1984 como un producto comercial, fue estandarizado por IEEE en 1995. Comenzó como una herramienta de descripción de circuitos lógicos pero rápidamente obtuvo capacidad de simular los mismos. Cuando fue lo suficientemente popular obtuvo la capacidad de sintetizar circuitos.
Luego de ser estandarizado en 1995 se hicieron mejoras y cambios que resultaron en un nuevo estándar IEEE1364-2001. Esta es la versión más soportada por las diversas herramientas. Luego en el año 2009 se une con SystemVerilog. Desde ese entonces todo cambio se publica bajo el lenguaje SystemVerilog.

Funciones booleanas

Función booleana F (3 variables)

$$F(A,B,C)=\overline{A}.B.C+A.\overline{B}.C+A.B.\overline{C}+A.B.C$$

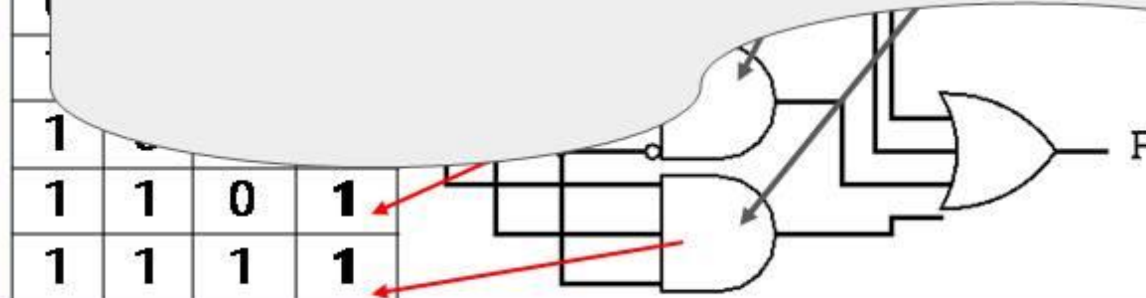
A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6

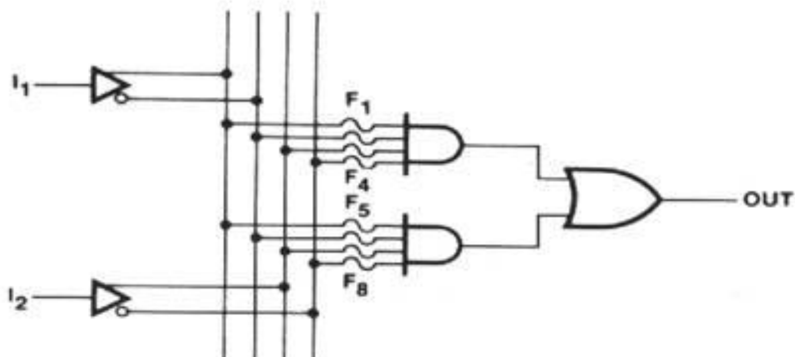
Función booleana F (3 variables)

Las tres formas de representación de la función (Tabla de verdad, circuito con compuertas o por definición de operaciones booleanas) SON EQUIVALENTES!. Es decir pasar de una forma a otra es completamente trivial. Las tres representan la misma función lógica.

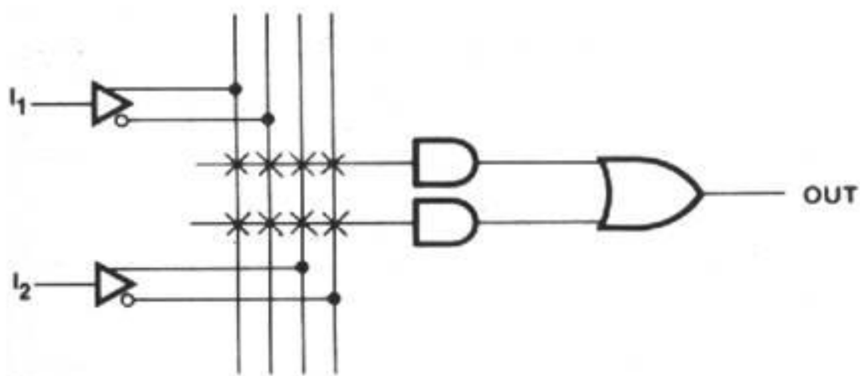


PAL

Circuito lógico real de una PAL

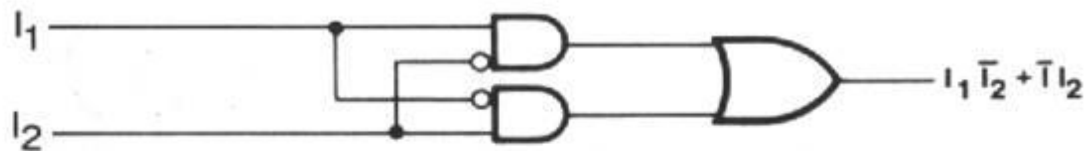


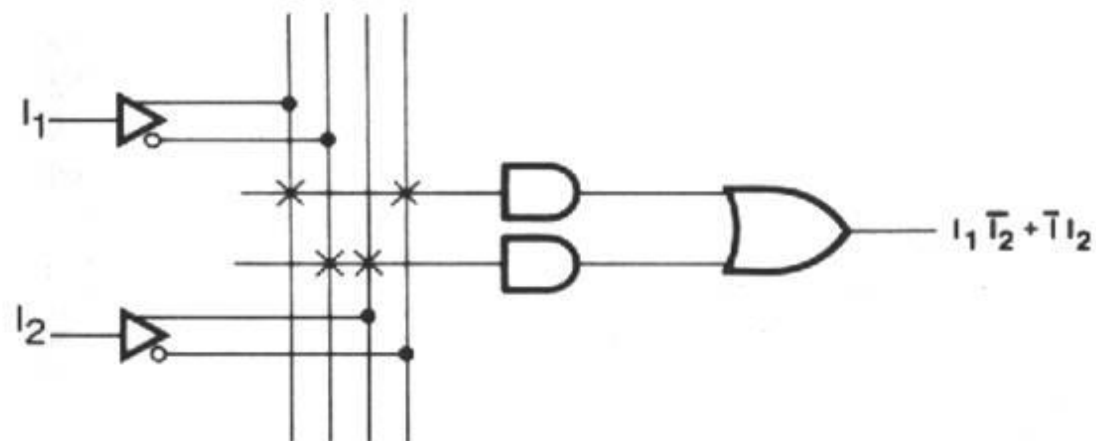
Circuito lógico simbólico de una PAL



Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6

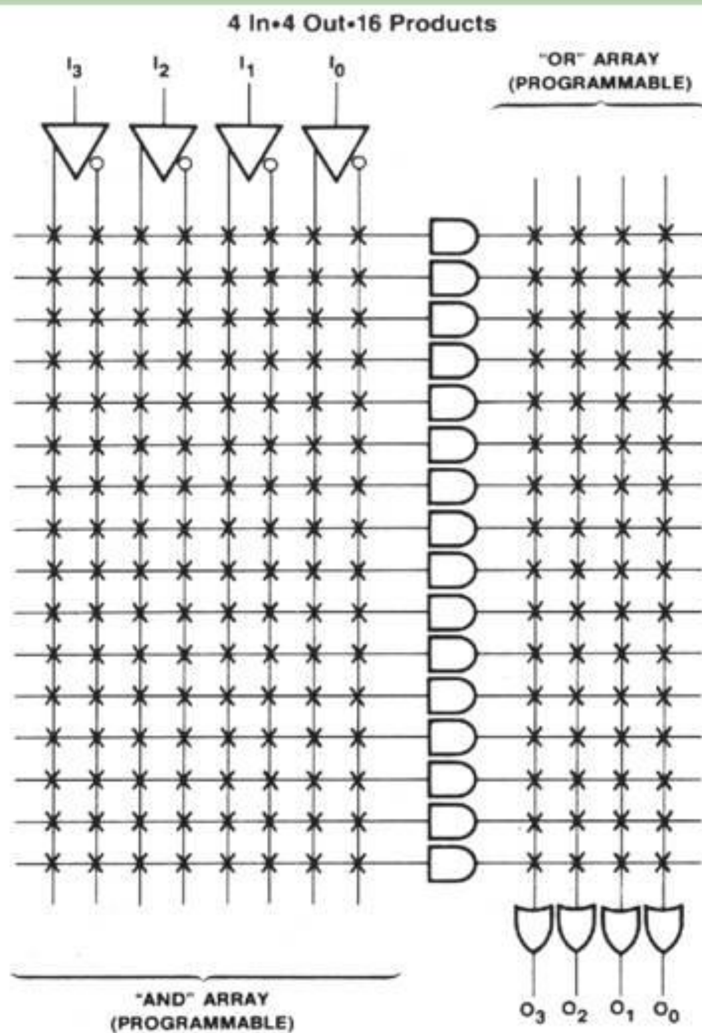
Implementación y esquema de quemado de fusibles





Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6

Pal de 4 entradas/salidas

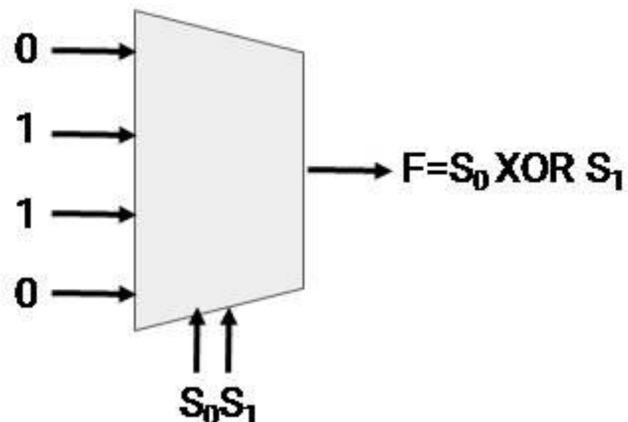


Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6

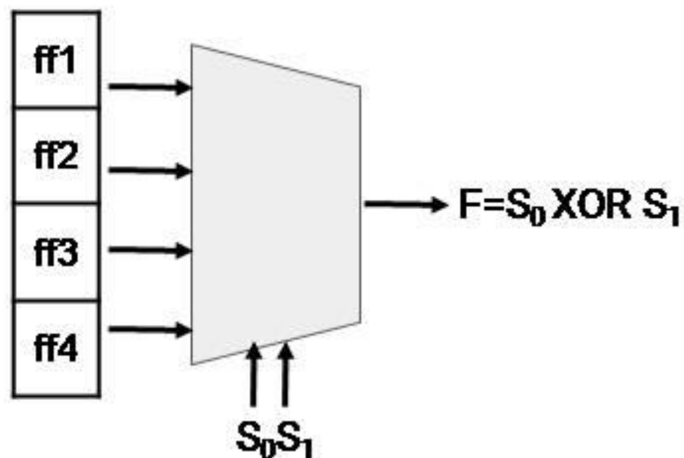
Funciones con multiplexores

Un MUX con entradas fijas

S_1	S_0	F
0	0	0
0	1	1
1	0	1
1	1	0



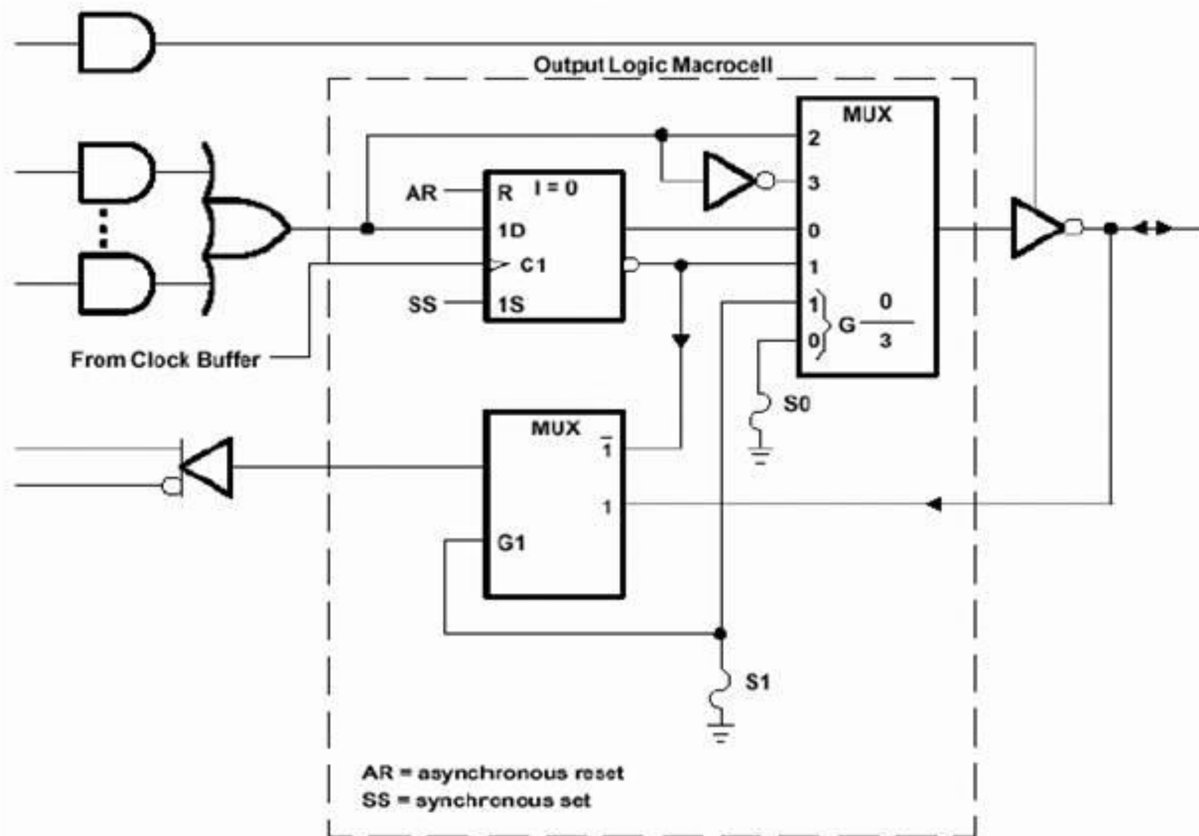
Podemos conectar un shift register compuesto de 4 FF a las entradas del MUX. Luego cambiando los valores de FFx podemos generar cualquier "compuerta" entre S0 y S1.



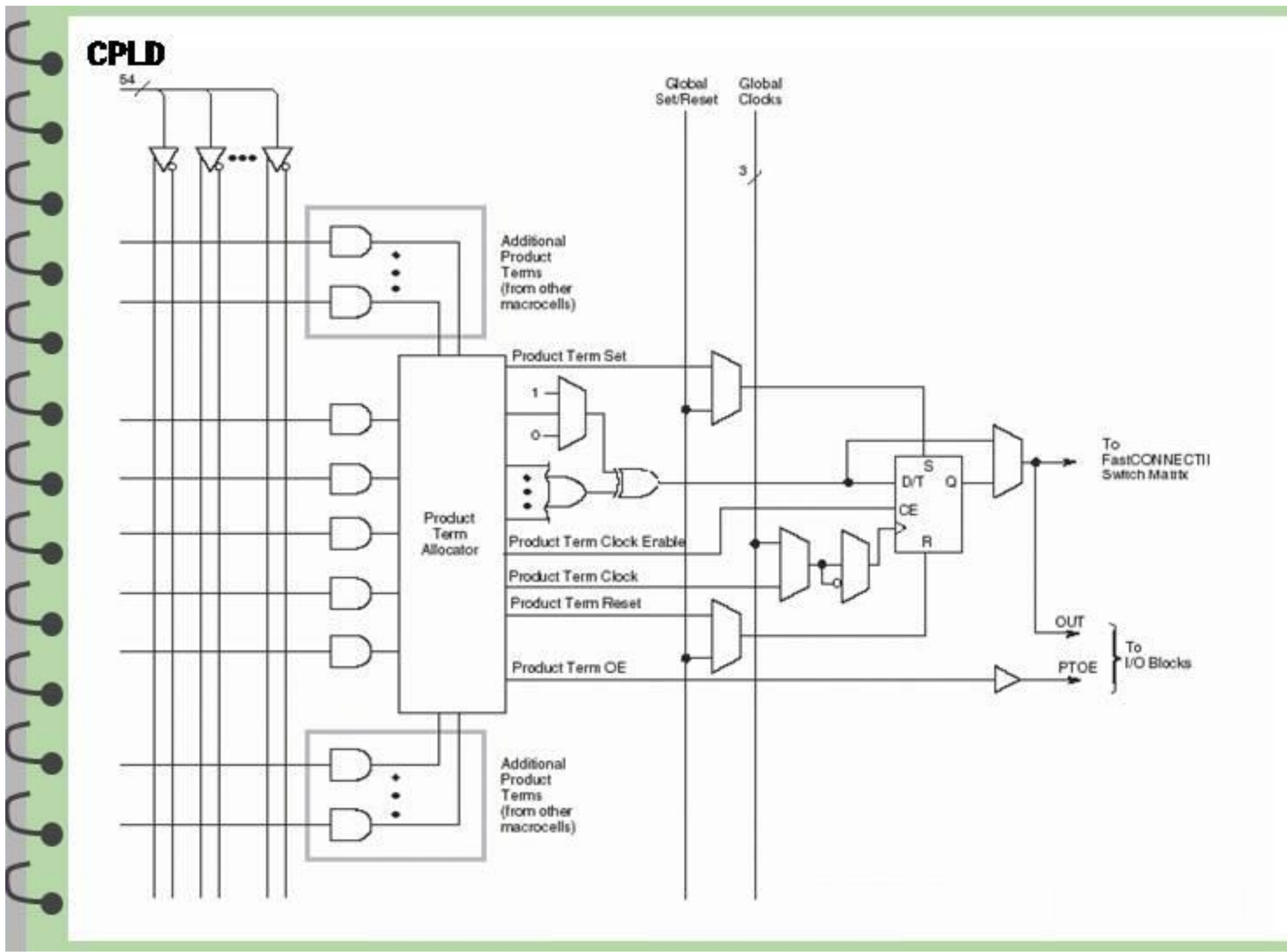
GAL y CPLD

Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6

Macrocelda de una GAL



Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6



Introducción 1

Módulos 2

Simulación 3

Secuenciales 4

Elementos Avanzados 5

Ejemplos 6

Curso de Verilog - Básico

FPGA CLB

Configurable Logic Block (CLB)

https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf

The 7 series configurable logic block (CLB) provides advanced, high-performance FPGA logic:

- Real 6-input look-up table (LUT) technology
- Dual LUT5 (5-input LUT) option
- Distributed Memory and Shift Register Logic capability
- Dedicated high-speed carry logic for arithmetic functions
- Wide multiplexers for efficient utilization

CLBs are the main logic resources for implementing sequential as well as combinatorial circuits. Each CLB element is connected to a switch matrix for access to the general routing matrix (shown in Figure 1-1). A CLB element contains a pair of slices.

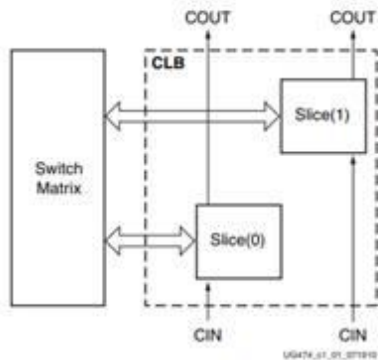
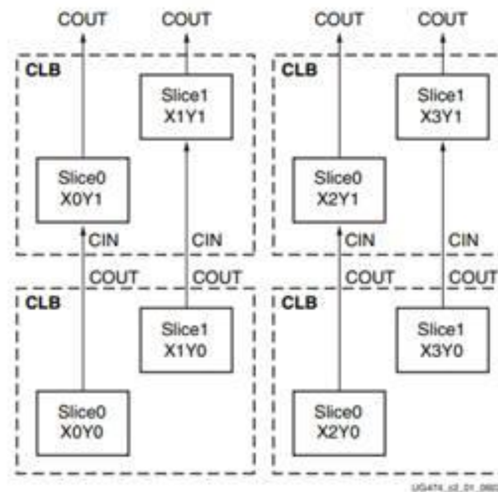


Figure 1-1: **Arrangement of Slices within the CLB**



Approximately two-thirds of the slices are SLICEL logic slices and the rest are SLICEM, which can also use their LUTs as distributed 64-bit RAM or as 32-bit shift registers (SRL32) or as two SRL16s. Modern synthesis tools take advantage of these highly efficient logic, arithmetic, and memory features. Expert designers can also instantiate them.

Configurable Logic Block (CLB)

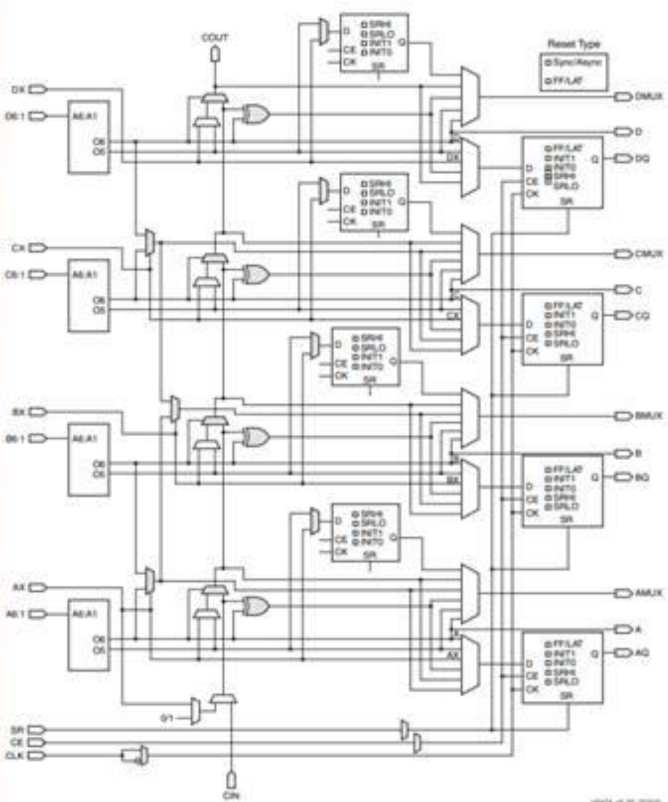


Figure 2-4: **Diagram of SLICEL**

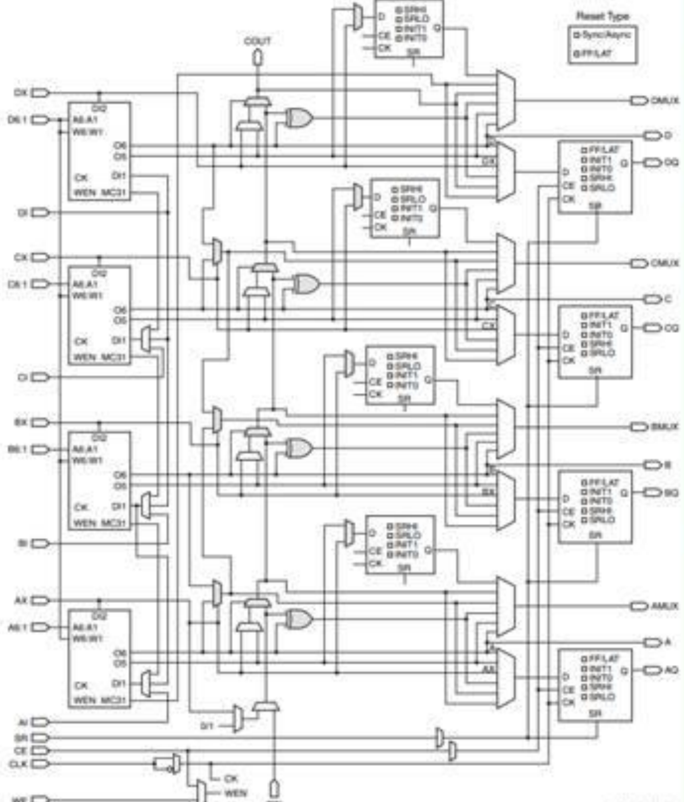


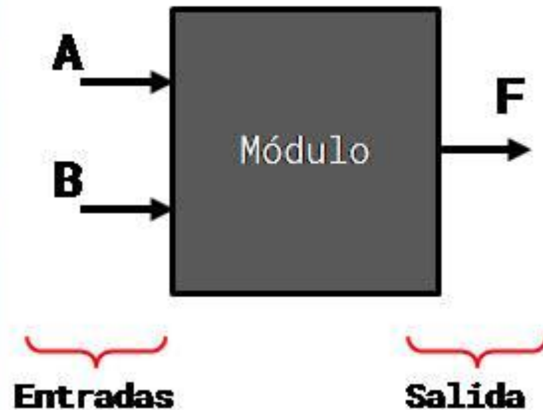
Figure 2-3: **Diagram of SLICEM**

Módulos en Verilog

Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6

Curso de Verilog - Básico

Diseño modular (Caja negra)



Entradas Salida

module *nombre (lista de puertos);*

- Interface
  - *Tipos de los puertos*
  - *Parámetros*
- Body
  - *Variables*
  - *Asignaciones*
  - *Instanciación de módulos*
  - *Bloques de simulación*
  - *Bloques de comportamiento*
  - *Funciones y tareas*

endmodule

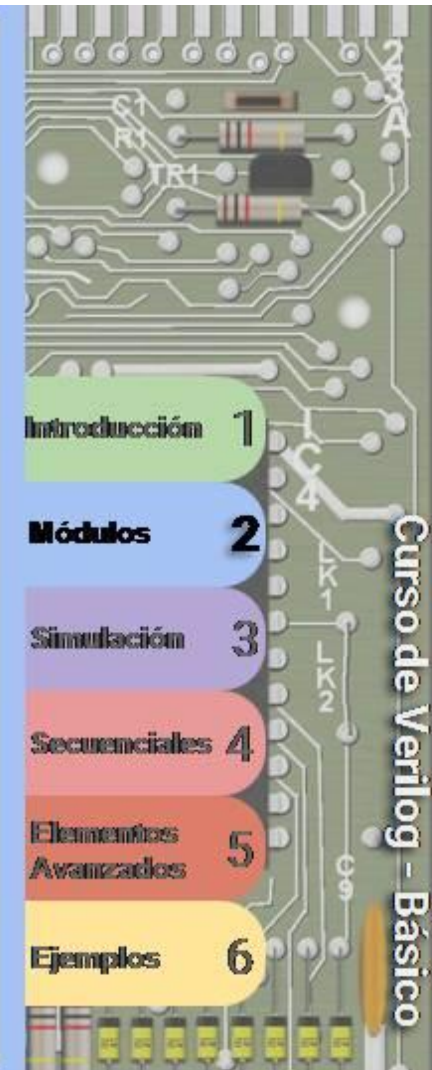
Verilog

```
module Modulo( A,B,F);
   input A;
   input B;
   output F;
endmodule
```

Verilog 2001

```
module Modulo(
    input A,
    input B,
    output F
    );
endmodule
```

```
module Modulo( input A,B,
               output F );
endmodule
```



Vivado (creando un módulo)

Sources

Project Summary

Overview | Dashboard

Add Sources

VIVADO ML Editions

Add Sources

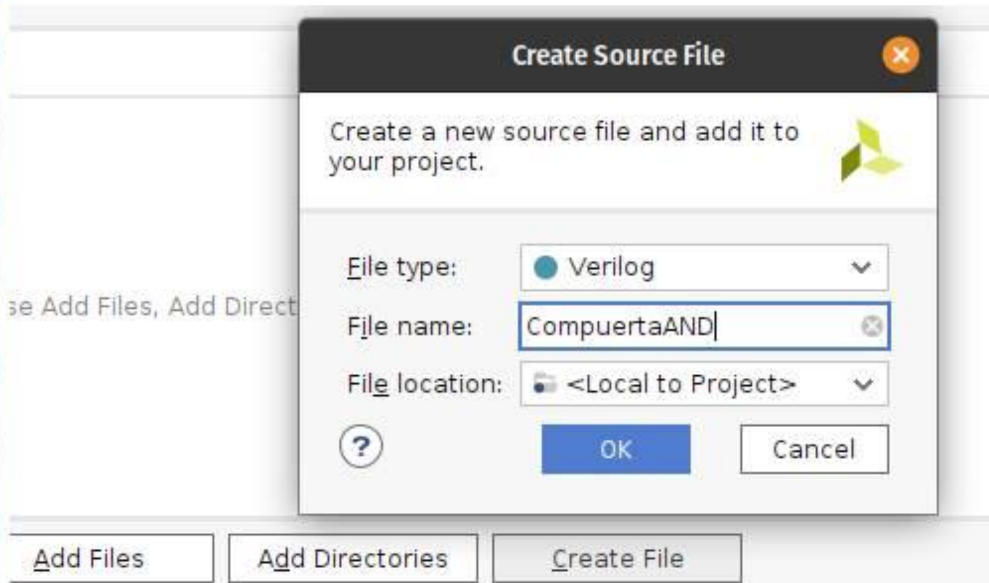
This guides you through the process of adding and creating sources for your project

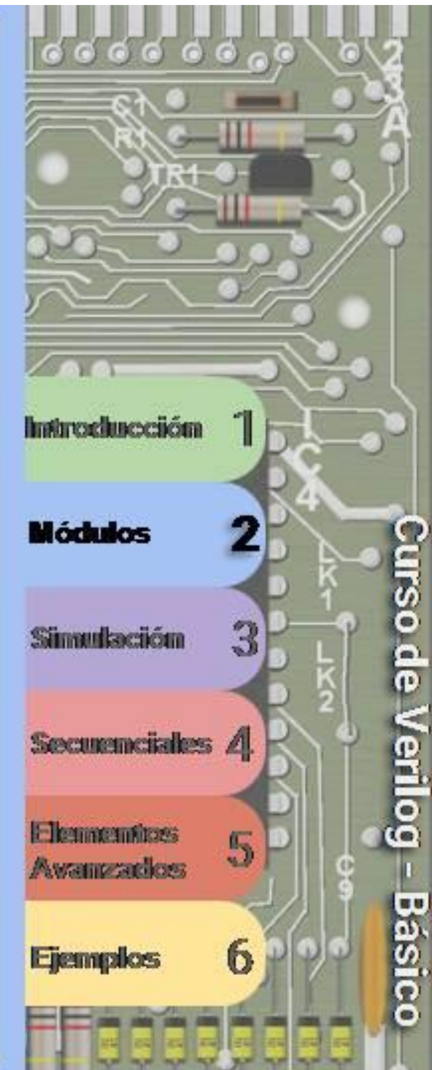
- ( ) Add or create constraints
- (•) Add or create design sources
- ( ) Add or create simulation sources

XILINX

< Back | Next > | Finish

Vivado (creando un módulo)





Vivado (creando un módulo)

Product family: Artix-7
Project part: xc7a35tcsg324-3
Top module name: TestModule
Target language: Verilog

Define Module

Define a module and specify I/O Ports to add to your source file.
For each port specified:
MSB and LSB values will be ignored unless its Bus column is checked.
Ports with blank names will not be written.

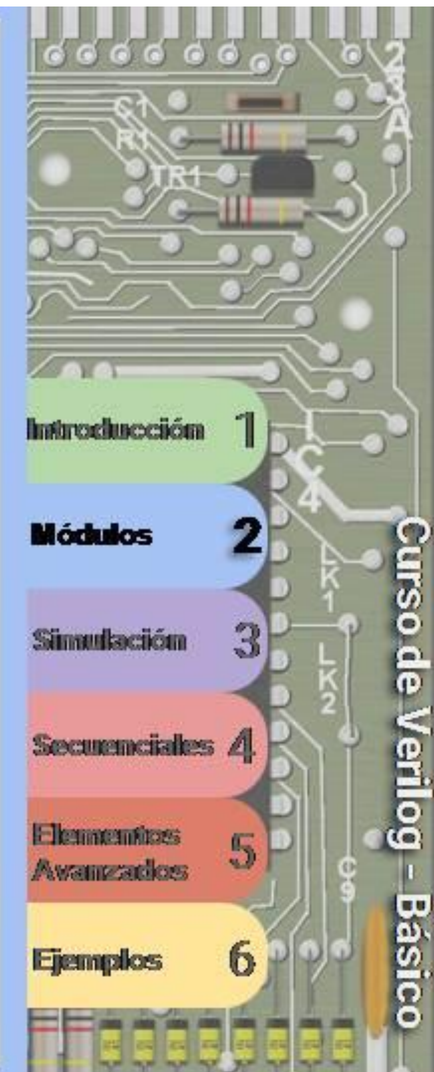
Module Definition

Module name: CompuertaAND

I/O Port Definitions

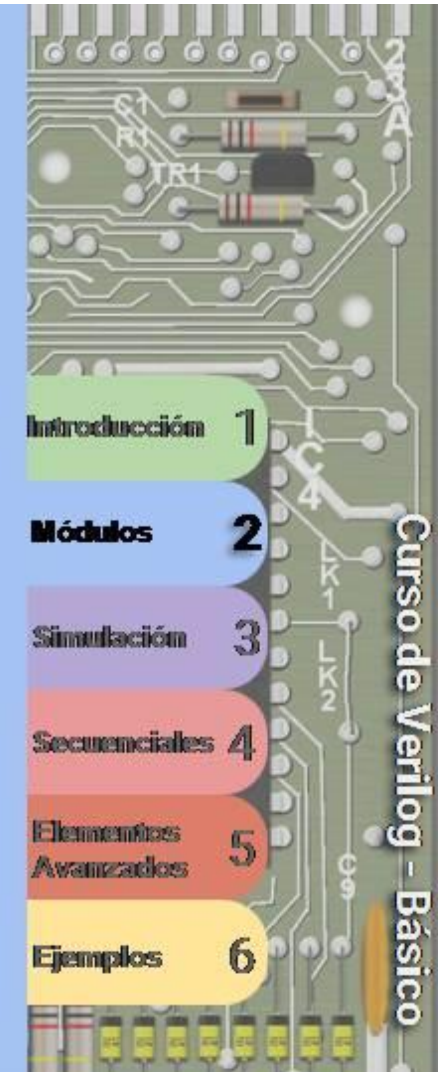
Port Name	Direction	Bus	MSB	LSB
A	input	☐	0	0
B	input	☐	0	0
F	output	☐	0	0

OK Cancel



Vivado (creando un módulo)

Puertos y Tipos de dato



Puertos y tipos de dato (verilog 2001)

Puertos
- **input**
- **output**
- **inout**

Por defecto los puertos son de tipo wire. Los output pueden definirse como **'reg'** (variable). Se puede asignar un rango **[x:y]** a un puerto para indicar un bus. Por defecto los buses se consideran como **'unsigned'** pero puede definirse como **'signed'** de ser necesario.

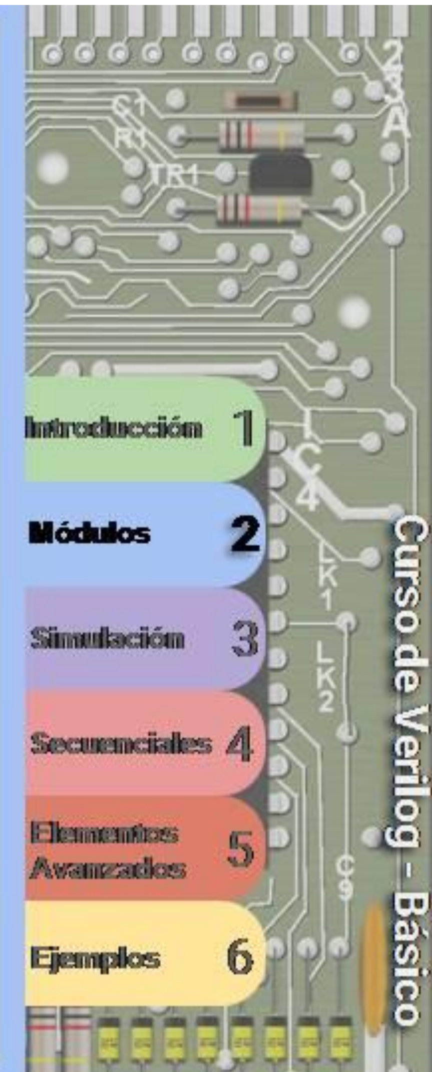
Valores posibles (wire/reg)
- **0**
- **1**
- **X (unknown)**
- **Z (High Impedance)**

Tipos de datos
- **wire**
- **reg**
- *integer*
- *time*
- *real*
- *string*

Algunos valores sólo tienen sentido en simulaciones.

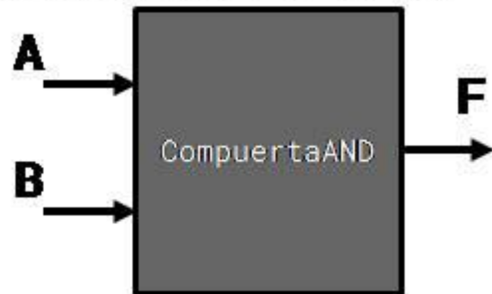
Valores literales

Se forman indicando la cantidad de bits, luego comilla simple y la base seguido del número. Ej: 4 bits (0110) se define como:
- **4'b0110**
- **4'h6**



Lógica combinatoria

Compuerta AND (assign)



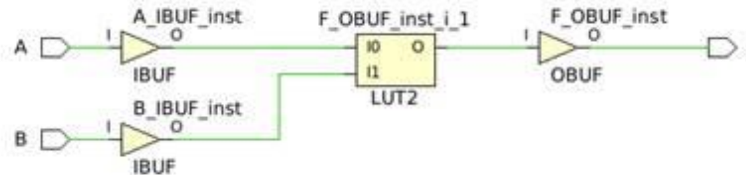
Entradas Salida

```
module CompuertaAND (
        input A,
        input B,
        output F
        );
    assign F = A & B;
endmodule
```

Operadores	
Símbolo	**Operación**
! ~	Negación (Bit)
&	And (Bit)
~&	Nand (Reduc)
|	Or (Bit)
~|	Nor (Reduc)
^	Xor (Bit)
~^	Xnor (Bit)

Vivado (Síntesis y esquemático)

- SYNTHESIS
  - Run Synthesis
  - Open Synthesized Design
    - Constraints Wizard
    - Edit Timing Constraints
    - Set Up Debug
    - Report Timing Summary
    - Report Clock Networks
    - Report Clock Interaction
    - Report Methodology
    - Report DRC
    - Report Noise
    - Report Utilization
    - Report Power
    - Schematic



https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/7series_scm.pdf

Cell Properties

F_OBUF_inst_i_1

CLASS	cell
FILE_NAME	/home/edgardog/Ne
INIT	4'h8

LUT2

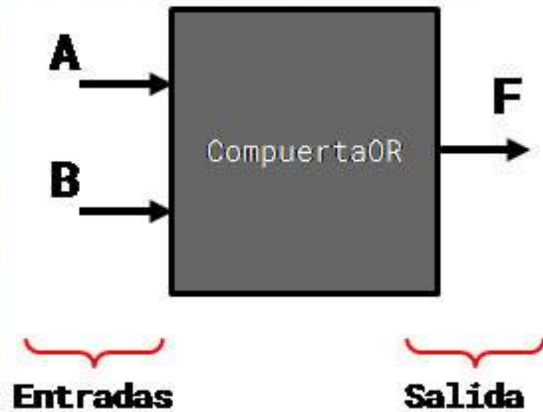
Primitive: 2-Bit Look-Up Table with General Output

Logic Table

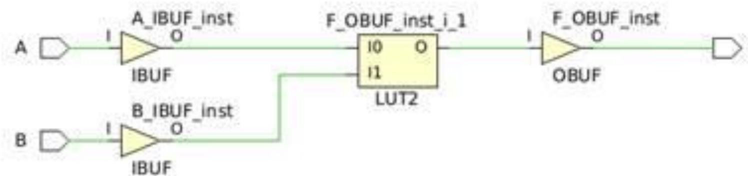
Inputs		Outputs
I1	I0	O
0	0	INIT[0]
0	1	INIT[1]
1	0	INIT[2]
1	1	INIT[3]

INIT = Binary equivalent of the hexadecimal number assigned to the INIT attribute

Compuerta OR (assign)




```
module CompuertaOR(
    input A,
    input B,
    output F
    );
    assign F = A | B ;
endmodule
```



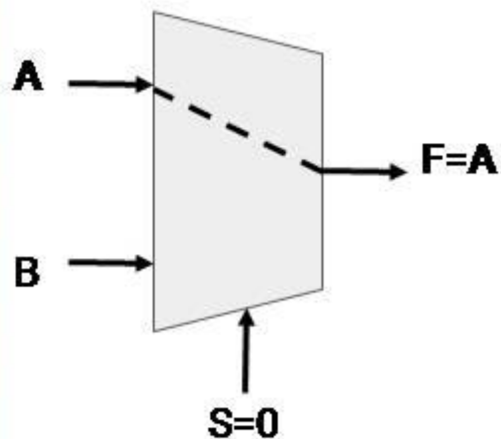
A	B	Or
0	0	0
0	1	1
1	0	1
1	1	1

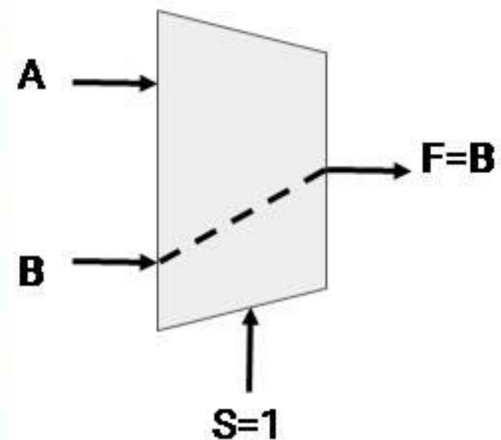
Cell Properties

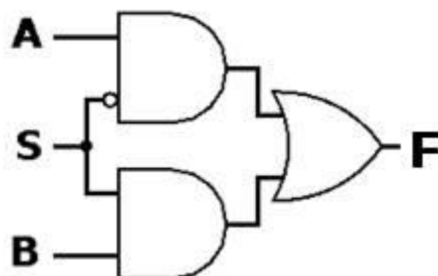
F_OBUF_inst_i_1

CLASS	cell
FILE_NAME	/home/edgardog/Ne
INIT	4'hE

Multiplexor (assign)

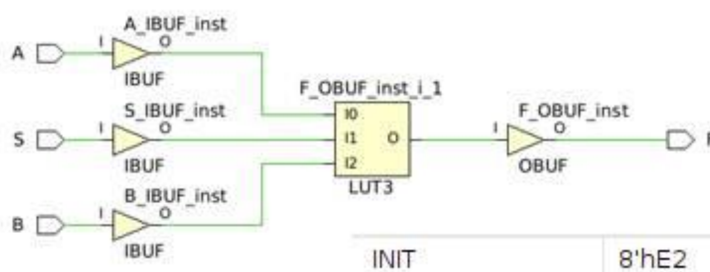






B	S	A	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

```
module Mux2(
    input A,
    input B,
    input S,
    output F
    );
    assign F = ( A & !S) | ( B & S);
endmodule
```

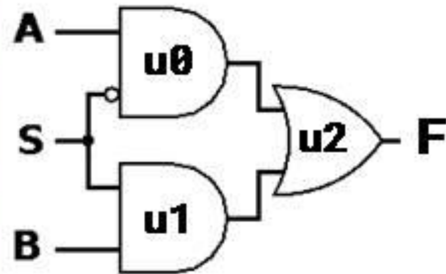


INIT	8'hE2

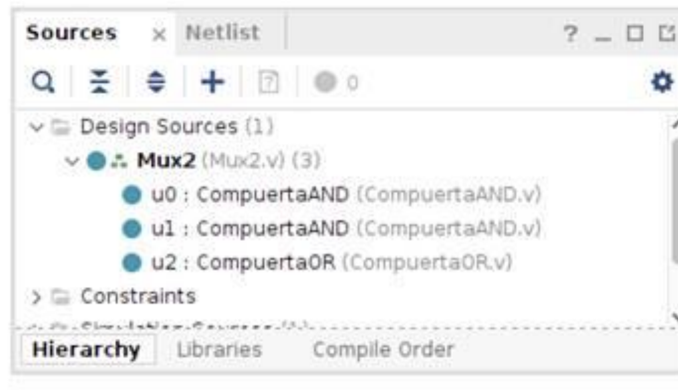
Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6

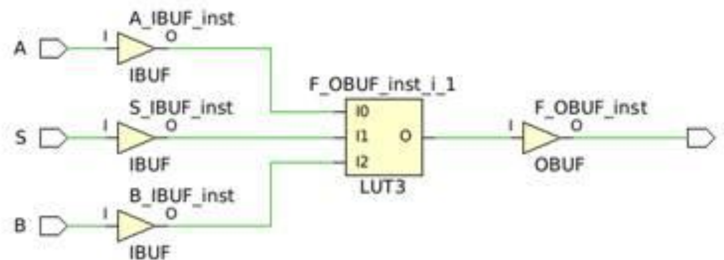
Curso de Verilog - Básico

Multiplexor (instancia de módulo)

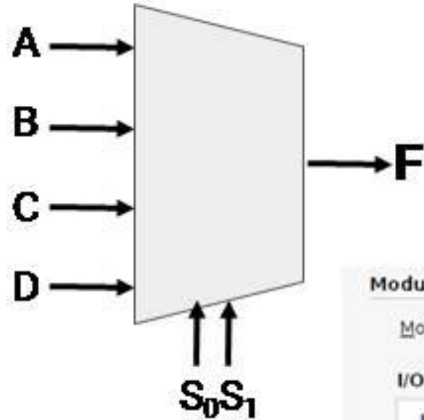


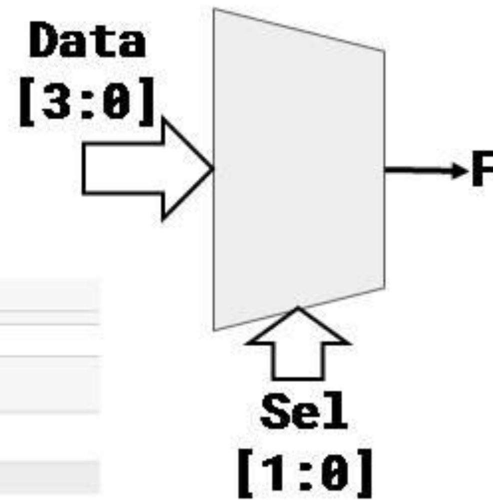

```
module Mux2(
    input A,
    input B,
    input S,
    output F
    );

    wire outAnd0;
    wire outAnd1;
    wire notS;

    assign notS = !S;
    //instancia definiendo nombres
    CompuertaAND u0 (.A(A),
                     .B(notS),
                     .F(outAnd0));
    //instancia ordenada
    CompuertaAND u1 (B,S,outAnd1);

    CompuertaOR u2 (outAnd0,outAnd1,F);
endmodule
```





Multiplexor (buses)





Module Definition

Module name: Mux4

I/O Port Definitions

Port Name	Direction	Bus	MSB	LSB
Data	input	☑	3	0
Sel	input	☑	1	0
F	output	☐	0	0

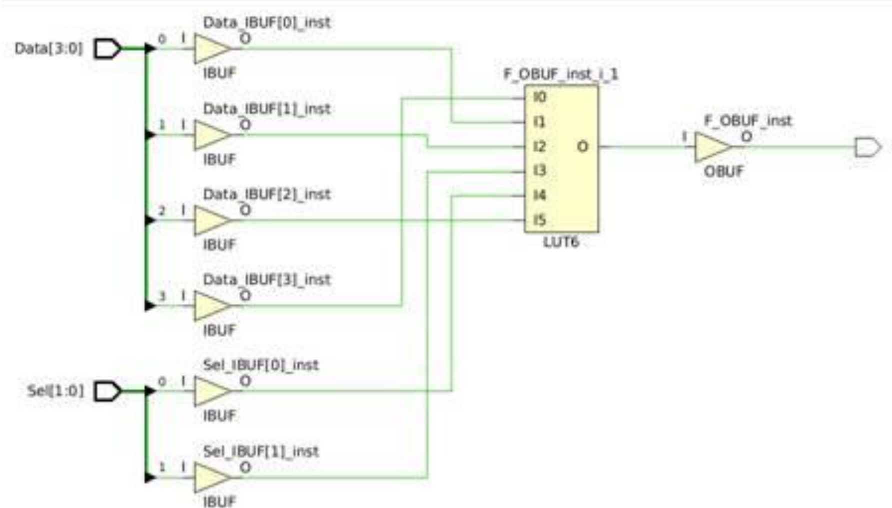
```
module Mux4(
    input [3:0] Data,
    input [1:0] Sel,
    output F
    );
endmodule
```

S_1	S_0	F
0	0	A
0	1	B
1	0	C
1	1	D

Sel	F
00	Data[0]
01	Data[1]
10	Data[2]
11	Data[3]

Multiplexor (buses)

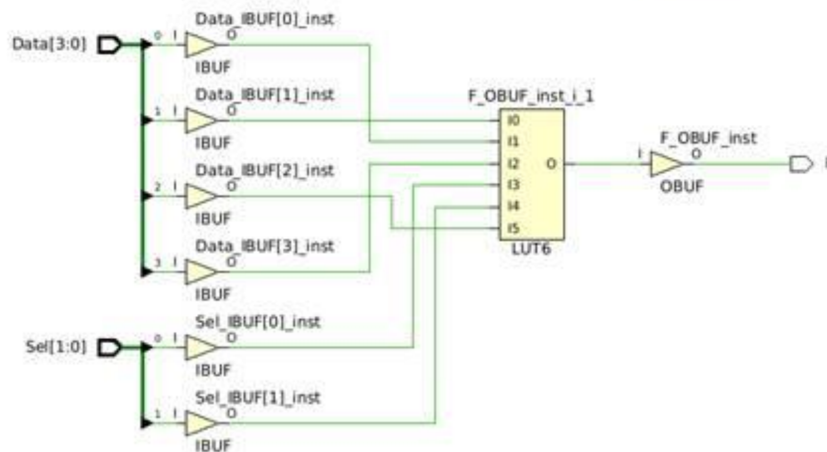
```verilog
module Mux4(
    input [3:0] Data,
    input [1:0] Sel,
    output F
    );

    assign F = ( Data[0] & !Sel[1] & !Sel[0] ) |
               ( Data[1] & !Sel[1] &  Sel[0] ) |
               ( Data[2] &  Sel[1] & !Sel[0] ) |
               ( Data[3] &  Sel[1] &  Sel[0] ) ;
endmodule
```



Multiplexor (adelanto de procesos)

```
module Mux4(
    input [3:0] Data,
    input [1:0] Sel,
    output reg F
    );

    always @*
    begin
        case(Sel)
            2'b00   : F = Data[0];
            2'b01   : F = Data[1];
            2'b10   : F = Data[2];
            default: F = Data[3];
        endcase
    end
endmodule
```

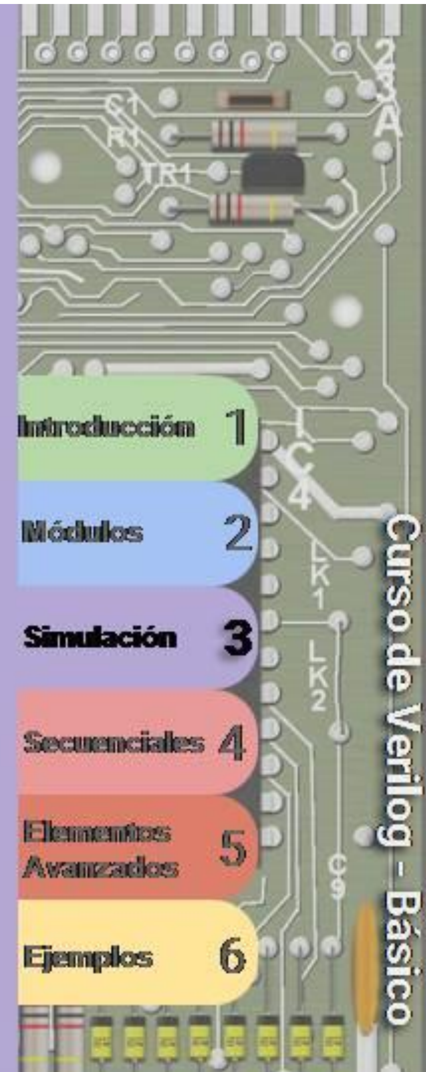
```
module Mux4(
    input [3:0] Data,
    input [1:0] Sel,
    output reg F
    );

    always @*
    begin
        if (Sel == 2'b00 )
            F = Data[0];
        else if (Sel == 2'b01)
            F = Data[1];
        else if (Sel == 2'b10)
            F = Data[2];
        else
            F = Data[3];
    end
endmodule
```



Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6

Curso de Verilog - Básico

Simulando un módulo



Testbench

Add Sources

This guides you through the process of adding and creat

- ( ) Add or create constraints
- ( ) Add or create design sources
- (•) Add or create simulation sources

Specify simulation set: sim_1

Create Source File

Create a new source file and add it to your project.

File type: Verilog

File name: CompuertaAND_tb

File location: <Local to Project>

? OK Cancel

Scan ar

Module Definition

Module name: CompuertaAND_tb

I/O Port Definitions

Port Name	Direction	Bus	MSB	LSB
	input	☐	0	0

? OK Cancel

Testbench

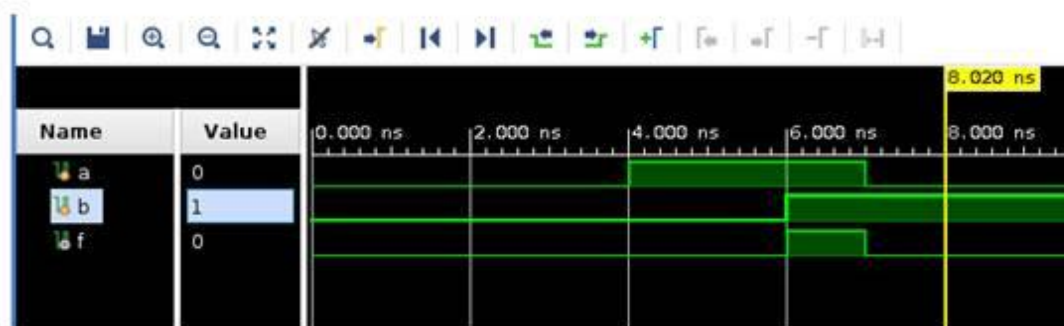
```
`timescale 1ns / 1ps

module CompuertaAND_tb();

    reg a,b;
    wire f;

    CompuertaAND U0 ( .A(a), .B(b), .F(f) );

    initial
        begin

        a=0;
        b=0;
        #4;

        a=1;
        #2;

        b=1;
        #1;

        a=0;
        #5;

        end
endmodule
```

Sources

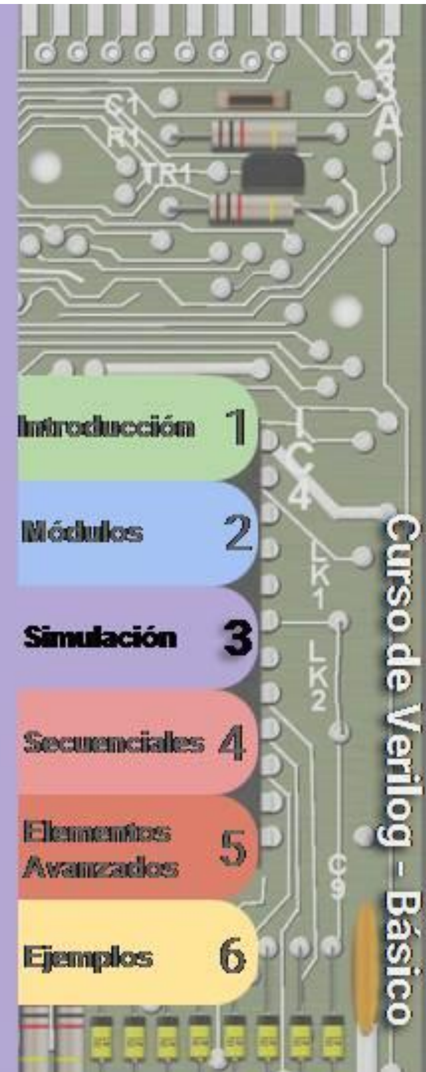
- Design Sources (2)
- Constraints
- Simulation Sources (3)
  - sim_1 (3)
    - CompuertaAND_tb (CompuertaAND_tb.v) (1)

SIMULATION

Run Simulation



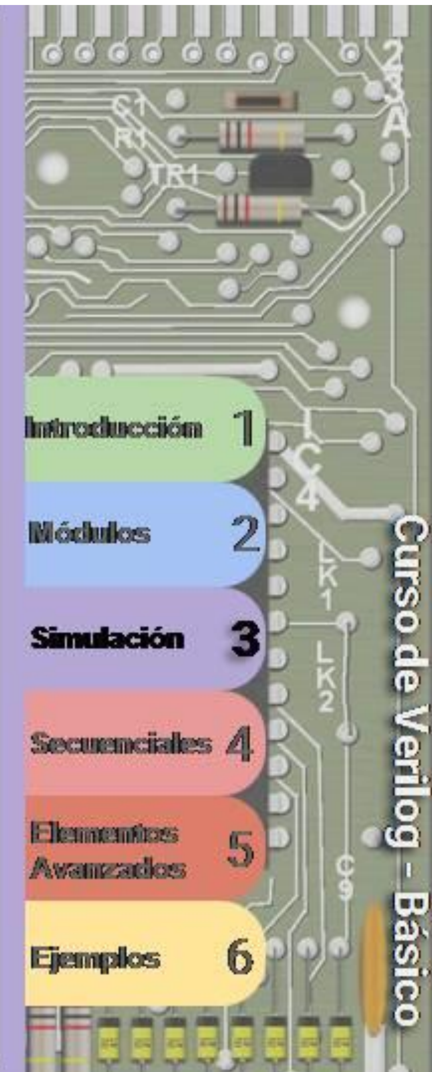
Funciones auxiliares



Testbench (archivo de log)

```
module CompuertaAND_tb();

    reg a,b;
    wire f;

    CompuertaAND U0 ( .A(a), .B(b), .F(f) );

    initial begin
        $monitor("Time=%0t , (A[%0d] AND B[%0d]) = F[%0d]",
                  $time,a,b,f);
    end

    initial
        begin

        a=0;
        b=0;
        #4;

        a=1;
        #2;

        b=1;
        #1;

        a=0;
        #5;

        end
endmodule
```

```
# run 1000ns
Time=0 , (A[0] AND B[0]) = F[0]
Time=4000 , (A[1] AND B[0]) = F[0]
Time=6000 , (A[1] AND B[1]) = F[1]
Time=7000 , (A[0] AND B[1]) = F[0]
```



Testbench (funciones auxiliares)

```
module CompuertaAND_tb();

    reg a,b;
    wire f;

    CompuertaAND U0 ( .A(a), .B(b), .F(f) );

    initial
        begin

        a=0;
        b=0;
        #4;

        a=1;
        #2;
        $display("Time=%0t A=%d",$time,a);
        b=1;
        #1;

        a=0;
        #5;

        end
endmodule
```

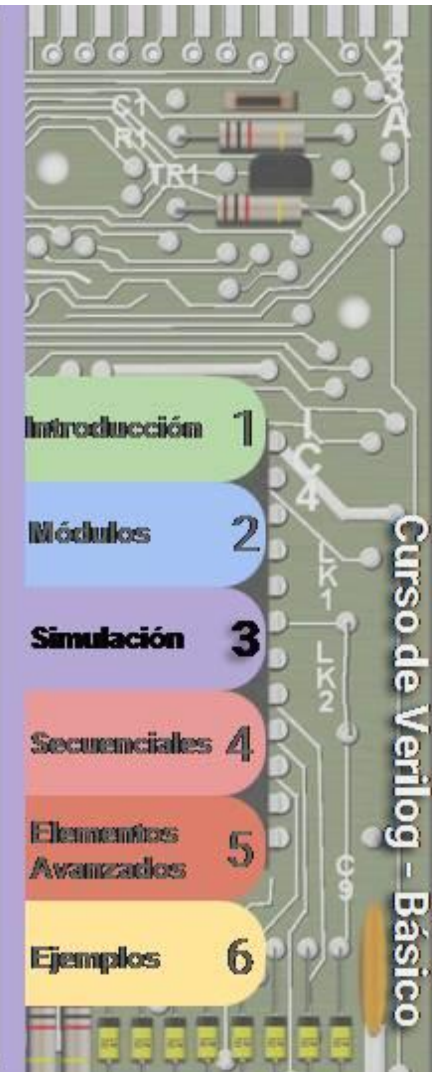
```
# run 1000ns
Time=6000 A=1
```

```
//Display inserta CRLF
 $display("Time=%0t A=%d",$time,a);
 //Write NO inserta CRLF
 $write("Time=%0t A=%d",$time,a);
 //Strobe muestra los valores ANTERIORES a $time
 //mientras que $display muestra POSTERIORES
 $strobe("Time=%0t A=%d",$time,a);
```

Formateadores

- **%t Tiempo**
- **%h Hexadecimal**
- **%d Decimal**
- **%b Binario**
- **%s String**
- **%f Float**
- **%e Exponencial**

Existen funciones $f{open,close,monitor,display, write,strobe,gets,eof} que interactúan con archivos. $sformat es análoga a fprintf.

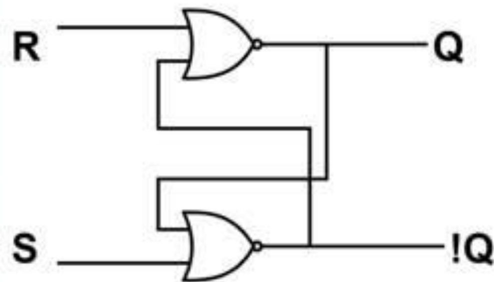


Flip Flops

Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6

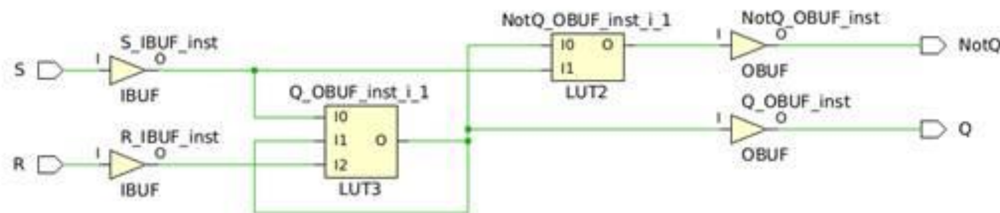
Curso de Verilog - Básico

Latch RS (NOR)




```
module CompuertaNOR(
    input A,
    input B,
    output F
    );

    assign F = ~|{A,B};
    //La operacion NOR reduce un bus.
    //No es bitwise (entre dos bits)
    //Usando { } se concatenan entradas
    //y luego ~| reduce a un bit
endmodule
```

```
module LatchRS(
    input R,
    input S,
    output Q,
    output NotQ
    );
    wire w1,w2;

   CompuertaNOR u0 ( .A(R),.B(w2),.F(w1));
   CompuertaNOR u1 ( .A(S),.B(w1),.F(w2));

   assign Q = w1;
   assign NotQ = w2;

endmodule
```



Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6

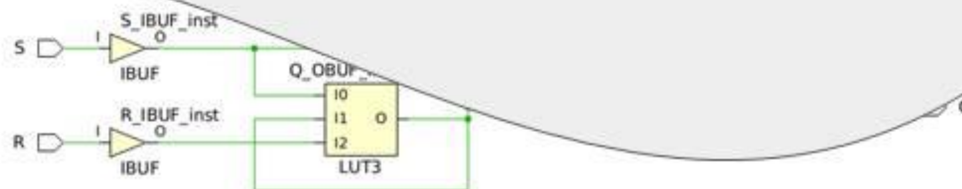
Latch RS (NOR)

```
module CompuertaNOR(
    input A,
    input B,
    output F
    );
```

```
~|{A,B};
```

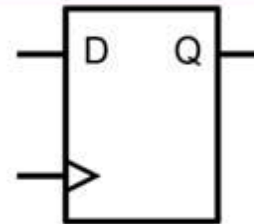
Implementación: **Combinatorial Loop Alert:** **1 LUT cells form a combinatorial loop**. **This can create a race condition**. Timing analysis may not be accurate. **The preferred resolution is to modify the design to remove combinatorial logic loops**. If the loop is known and understood, this DRC can be bypassed by..
One net in the loop is Q_OBUF. Please evaluate your design.
The cells in the loop are: Q_OBUF_inst_i_1.

```
NotQ = w2;

ndmodule
```

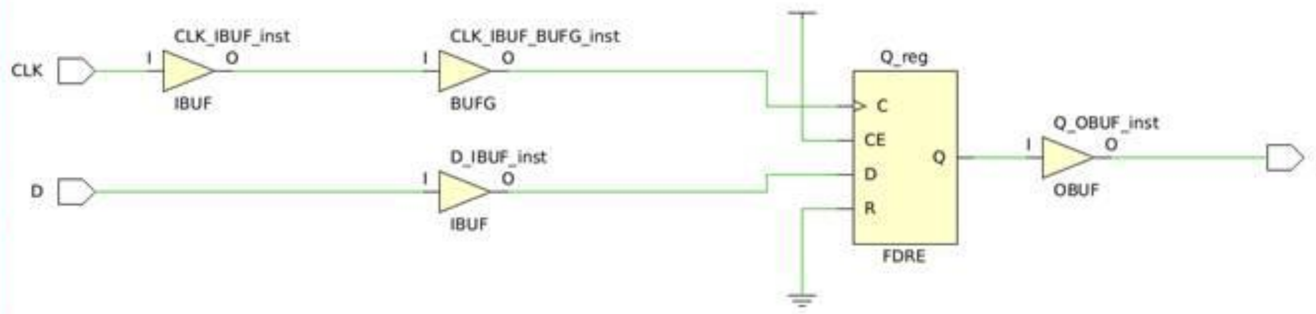


Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6
Curso de Verilog - Básico

Flip Flop D (flanco ascendente sin reset)



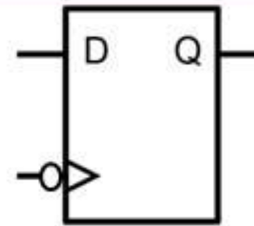

```
module FlipFlopD(
    input D,
    input CLK,
    output reg Q
    );

    always @(posedge CLK)
    begin
        Q <= D;
    end
endmodule
```



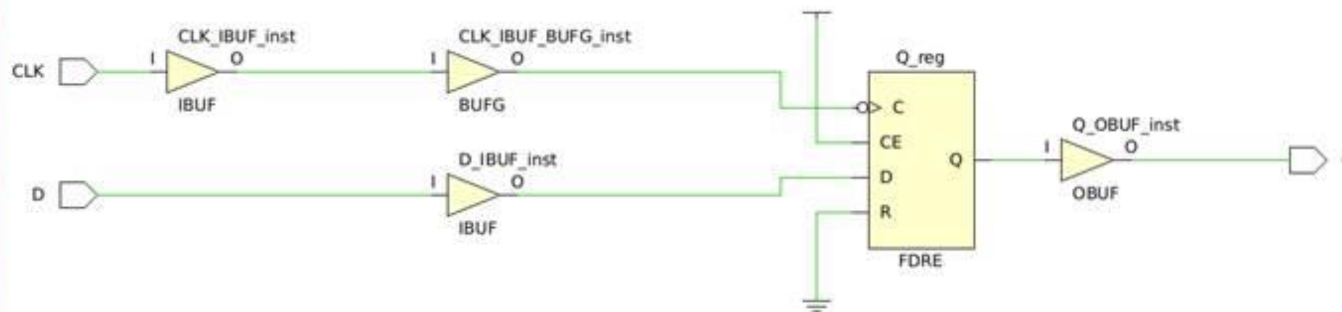
Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6

Curso de Verilog - Básico

Flip Flop D (flanco descendente sin reset)



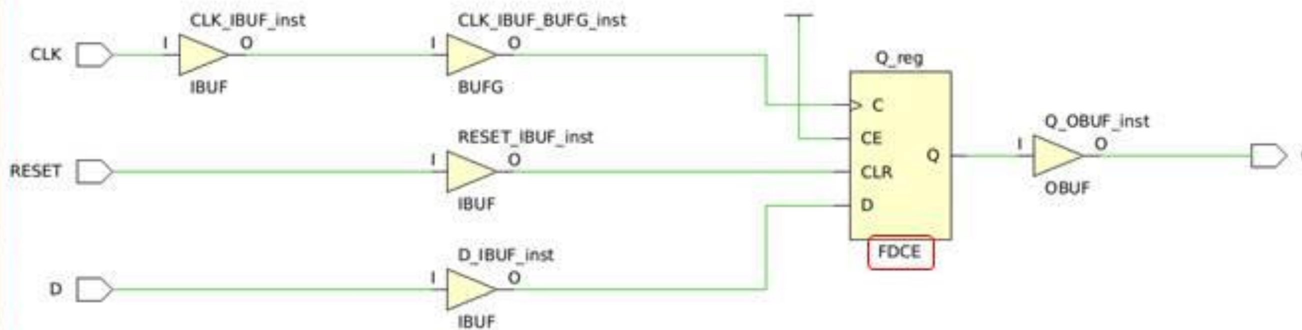

```
module FlipFlopD_desc(
    input D,
    input CLK,
    output reg Q
    );

    always @(negedge CLK)
    begin
        Q <= D;
    end
endmodule
```



Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6

Flip Flop D (Reset Asincrónico)

```
module FlipFlopD_Rasinc(
    input D,
    input CLK,
    input RESET,
    output reg Q
    );

    always @(posedge CLK or posedge RESET)
    begin
        if ( RESET==1'b1)
            Q <= 1'b0;
            else
            Q <= D;
    end
endmodule
```

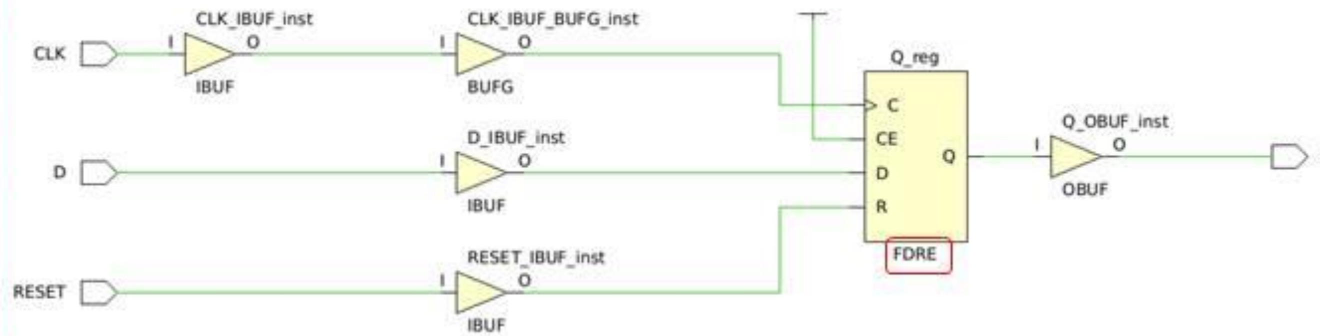


Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6

Curso de Verilog - Básico

Flip Flop D (Reset Sincrónico)

```
module FlipFlopD_Rsinc(
    input D,
    input CLK,
    input RESET,
    output reg Q
    );

    always @(posedge CLK)
    begin
        if ( RESET==1'b1)
            Q <= 1'b0;
            else
            Q <= D;
    end
endmodule
```



Introducción 1

Módulos 2

Simulación 3

Secuenciales 4

Elementos Avanzados 5

Ejemplos 6

Curso de Verilog - Básico

Flip Flop D (Simulando)

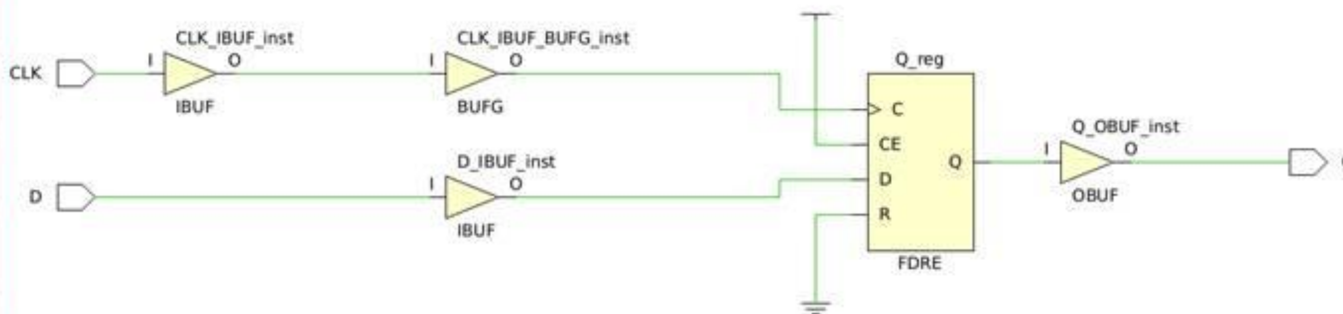
```
module FlipFlopD_tb();
reg D,CLK,RESET;
wire Q;
FlipFlopD_Rasinc U0(
        .D(D),
        .CLK(CLK),
        .RESET(RESET),
        .Q(Q) );

initial begin
    CLK=0;
    RESET=1;

    forever #5 CLK = !CLK;
end
```

```
initial begin
    D = 1;
    #20;
    D = 0;
    #20;
    RESET=0;
    #20;
    D = 1;
    #20;
    D = 0;
    #20;
    D = 1;
    #8;
    RESET=1;
    #12;
    D = 1;
    #20;
    D = 0;
end
endmodule
```

Asignaciones

Asignación bloqueante

```
module AsigBlock(
    input D,
    input CLK,
    output reg Q
    );

 reg aux;
 always @(posedge CLK)
 begin
    aux = D;
    Q = aux;
 end

endmodule
```



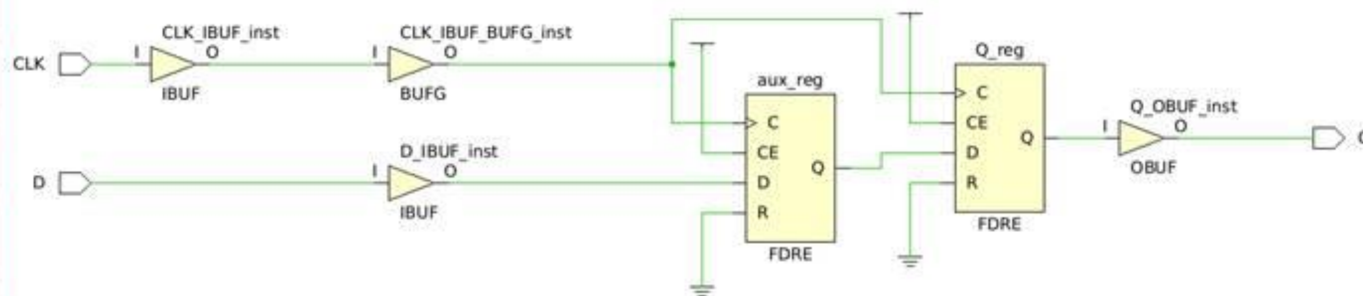
Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6

Curso de Verilog - Básico

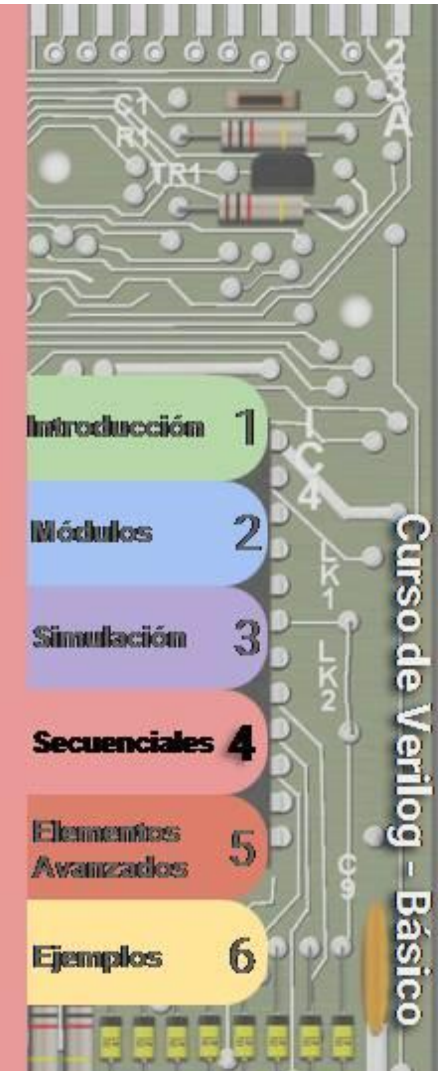
Asignación NO bloqueante

```
module AsigNoBlock(
    input D,
    input CLK,
    output reg Q
    );

 reg aux;
 always @(posedge CLK)
 begin
    aux <= D;
    Q <= aux;
 end

endmodule
```

```
module AsigNoBlock(
    input D,
    input CLK,
    output reg Q
    );

 reg aux;
 always @(posedge CLK)
 begin
    Q <= aux;
    aux <= D;
 end

endmodule
```

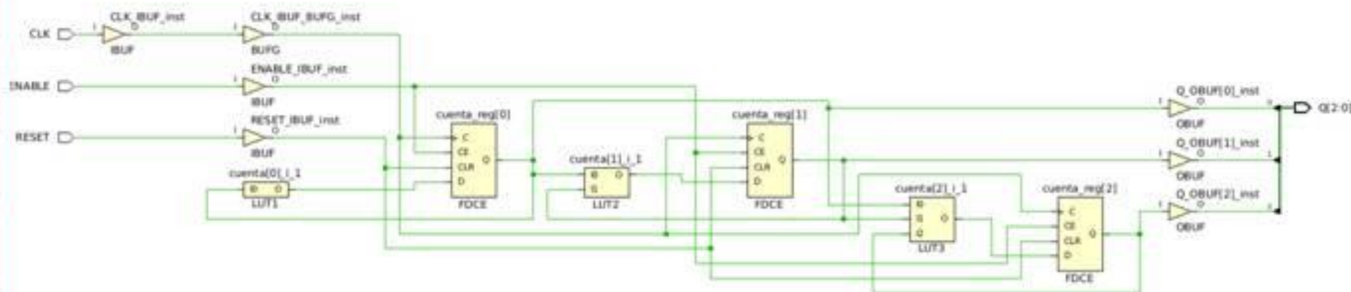


Contadores



Contador 3 bits (Ascendente)

```
module ContadorASC(
    input CLK,
    input RESET,
    input ENABLE,
    output [2:0] Q
    );
reg [2:0] cuenta;
assign Q = cuenta;

always @(posedge CLK or posedge RESET)
begin
    if (RESET)
        cuenta <= 3'd0;
    else
      begin
        if (ENABLE)
            cuenta <= cuenta + 3'd1;
      end
end
endmodule
```



Introducción 1

Módulos 2

Simulación 3

Secuenciales 4

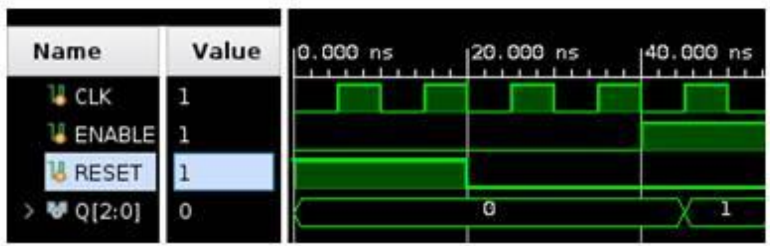
Elementos Avanzados 5

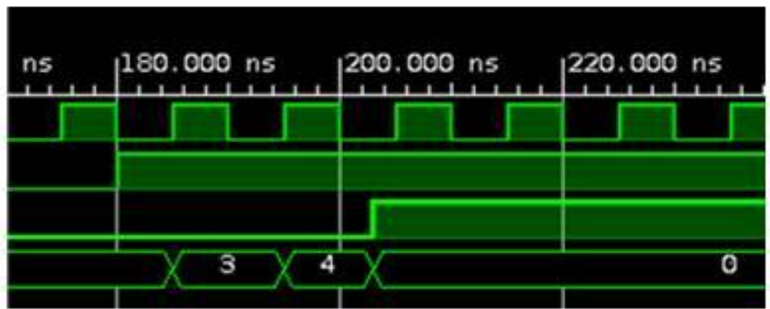
Ejemplos 6

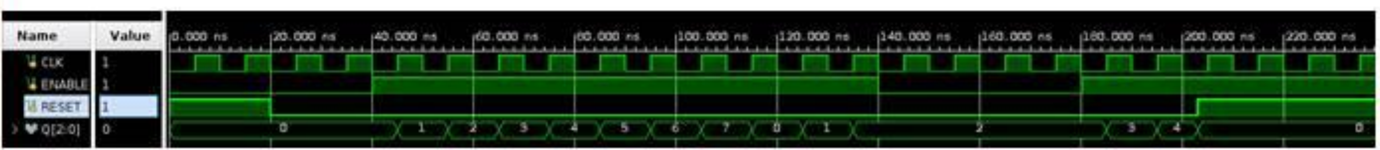
Curso de Verilog - Básico

Contador 3 bits (Simulación)

```
module ContadorASC_tb();
reg CLK,ENABLE,RESET;
wire [2:0] Q;

    ContadorASC U0 (CLK,RESET,ENABLE,Q);
initial begin
    CLK=0;
    ENABLE=0;
    RESET=1;
    forever #5 CLK=~CLK;
end

initial begin
    #20;
    RESET=0;
    #20;
    ENABLE=1;
    #100;
    ENABLE=0;
    #40;
    ENABLE=1;
    #23;
    RESET=1;
end
endmodule
```







Introducción 1

Módulos 2

Simulación 3

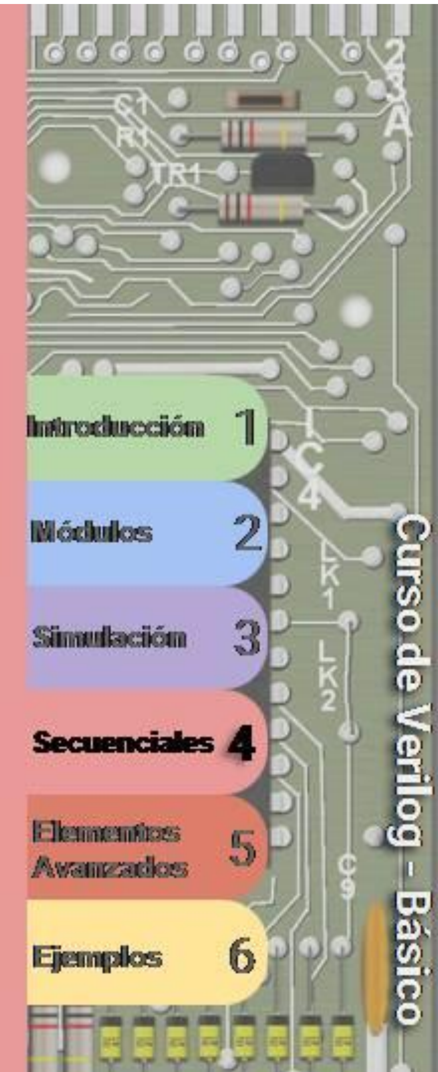
Secuenciales 4

Elementos Avanzados 5

Ejemplos 6

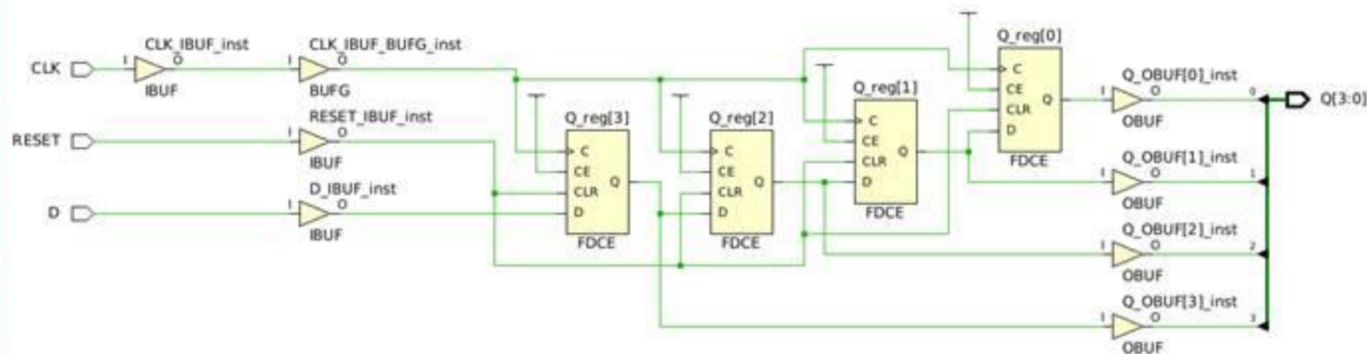
Curso de Verilog - Básico

Desplazamiento



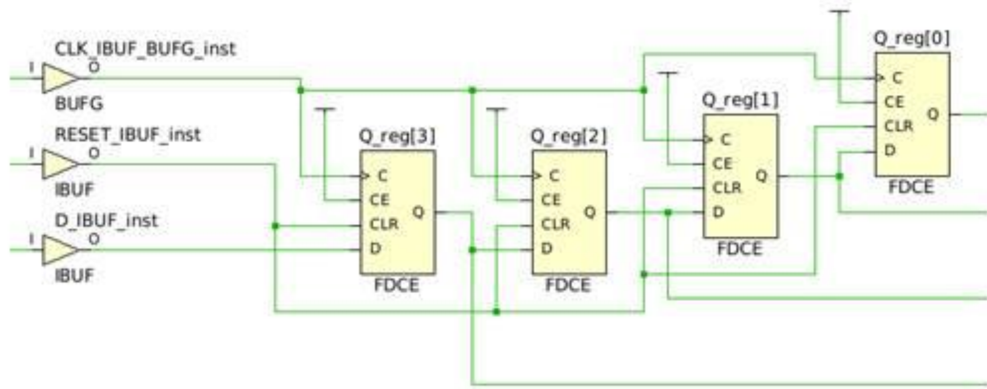
Shifter 4 bits

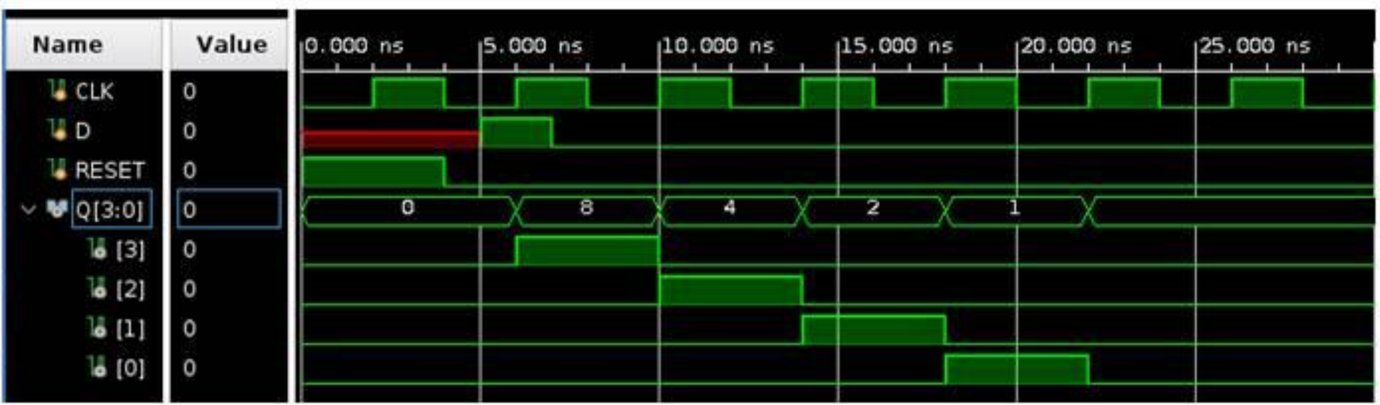
```
module ShiftRight(
    input D,
    input CLK,
    input RESET,
    output reg [3:0]  Q
    );

always @(posedge CLK or posedge RESET)
begin
     if (RESET)
        Q <= 0;
     else
      Q <= { D , Q[3:1]};

end
endmodule
```



Shifter 4 bits (testbench)

```
module ShiftRight_tb();
reg CLK,D,RESET;
wire [3:0] Q;

ShiftRight U0 (D,CLK,RESET,Q);
initial begin
    CLK=0;
    RESET=1;
    forever #2 CLK=~CLK;
end

initial begin
    #4;
    RESET=0;
    #1;
    D=1;
    #2;
    D=0;
end
endmodule
```





Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6

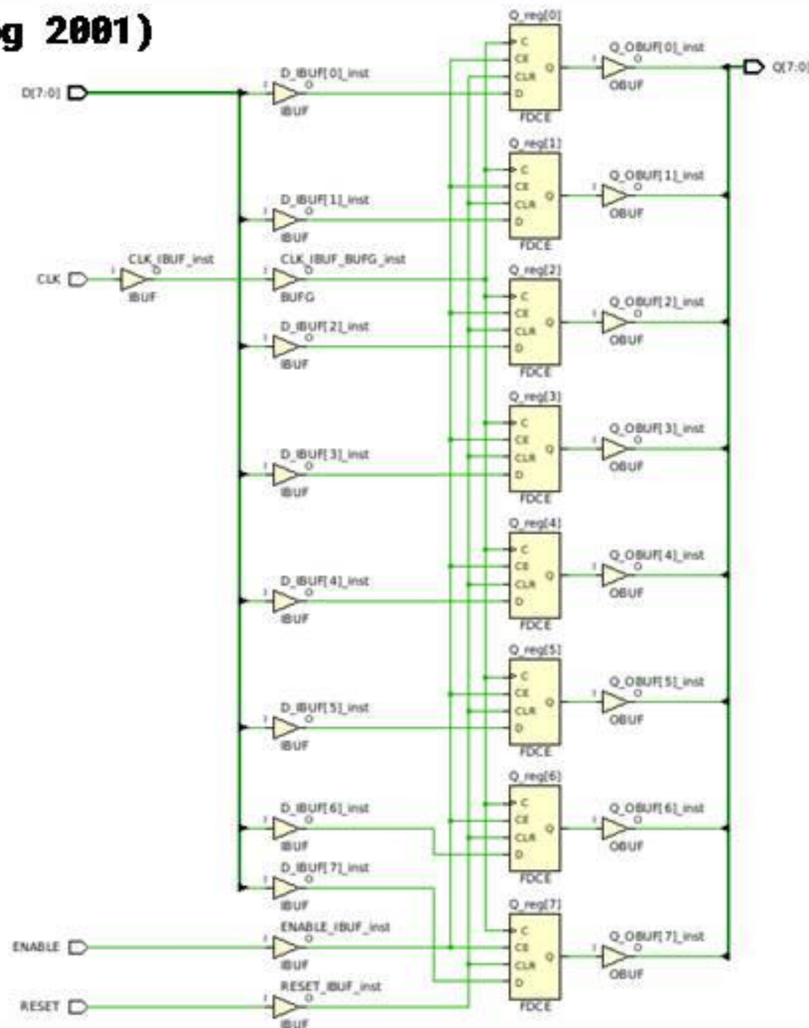
Parámetros

Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6

Curso de Verilog - Básico

Definir parámetro (Verilog 2001)

```
module LatchRegister
#(parameter SIZE=8)
(
    input [SIZE-1:0] D,
    input CLK,
    input RESET,
    input ENABLE,
    output reg [SIZE-1:0] Q
    );

always@(posedge CLK or posedge RESET)
begin
    if (RESET)
        Q <= 0;
    else
        if (ENABLE)
            Q <= D;
end

endmodule
```



Cambiar parámetro en instancia (Verilog 2001)

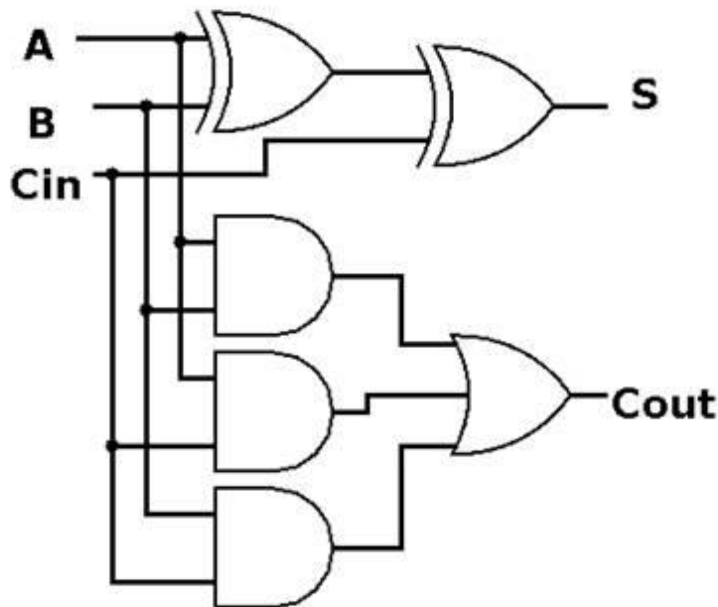
```
module LatchRegister_tb
#(parameter [3:0] initVal = 4'b0110)
();

reg CLK,ENABLE,RESET;
reg [3:0] D;
wire [3:0] Q;

    LatchRegister #(.SIZE(4) ) U0
        (D,CLK,RESET,ENABLE,Q);

 initial begin
    CLK=0;
    ENABLE=0;
    RESET=1;
    D=initVal;
    forever #2 CLK=~CLK;
end

initial begin
    #2;
    D=4'b1110;
    #2;
    RESET=0;
    #4;
    ENABLE=1;
    #8;
    D=4'b0011;
    #4;
    RESET=1;
end
endmodule
```

Instanciar elementos internos

Full Adder (LUT)

entrada			salida	
C_{IN}	A	B	C_{OUT}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

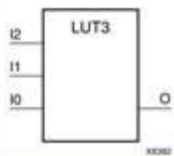
C_{OUT}=E8 S=96



https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/7series_hdl.pdf

LUT3

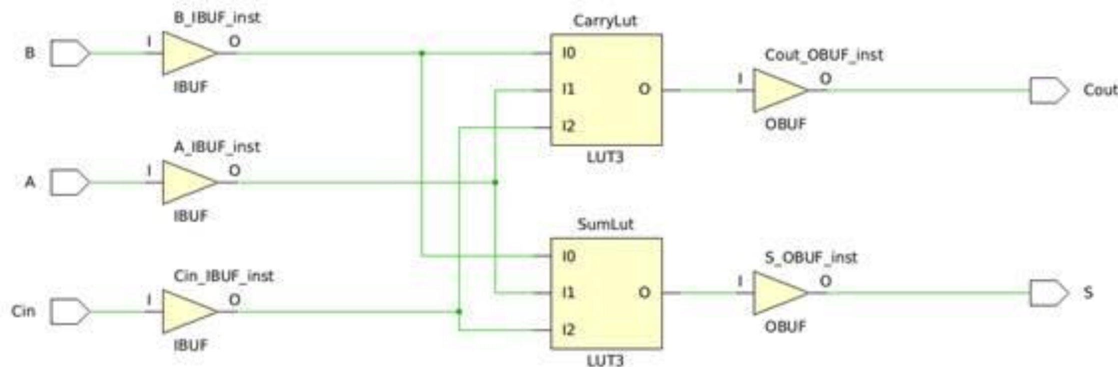
Primitive: 3-Bit Look-Up Table with General Output




```
LUT3 #(
   .INIT(8'h00)  // Specify LUT Contents
) LUT3_inst (
   .O(O),    // LUT general output
   .I0(I0), // LUT input
   .I1(I1), // LUT input
   .I2(I2)  // LUT input
);
```

Full Adder (LUT)

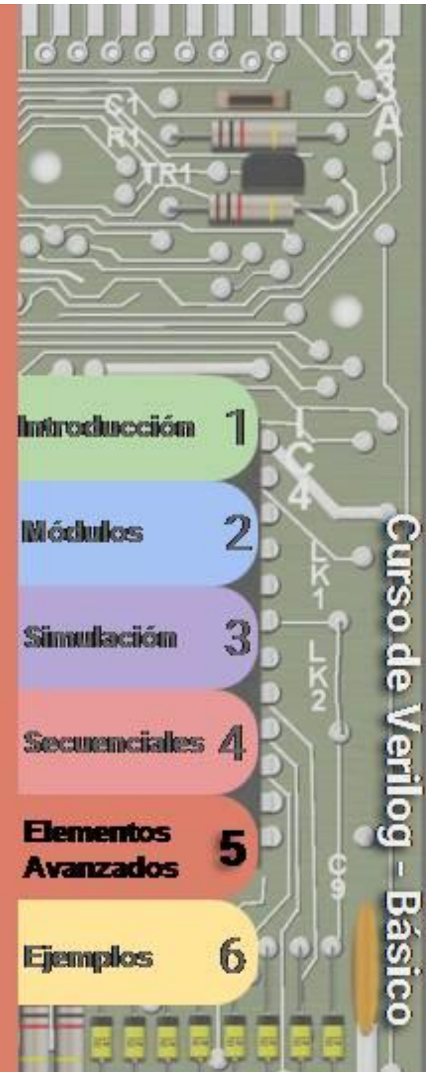
```
module FullAdderLut(
    input A,
    input B,
    input Cin,
    output S,
    output Cout
    );

LUT3  #(.INIT(8'hE8)) CarryLut (
    .O(Cout),
    .I0(B),
    .I1(A),
    .I2(Cin) );

LUT3  #(.INIT(8'h96)) SumLut (
    .O(S),
    .I0(B),
    .I1(A),
    .I2(Cin) );
endmodule
```



Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6

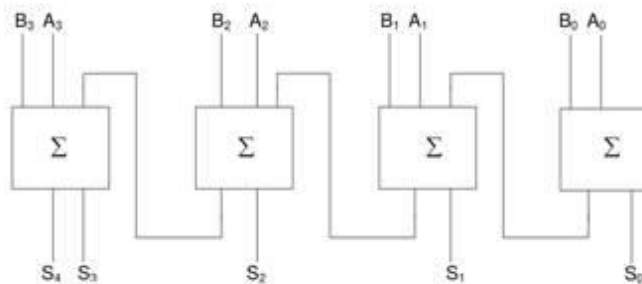
Curso de Verilog - Básico

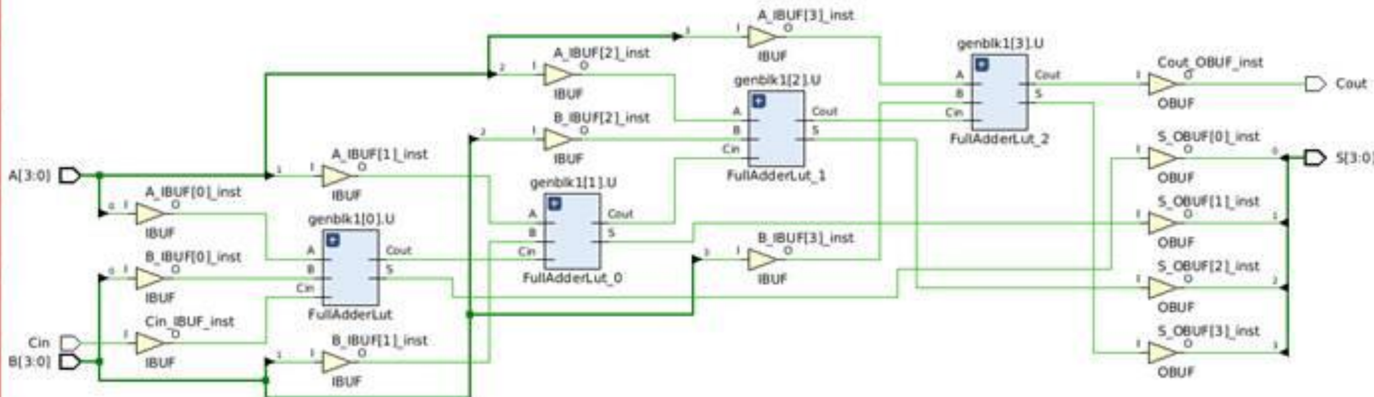
For Generate



Hex Adder (generate for)

```
module HexAdder(
    input [3:0] A,
    input [3:0] B,
    input Cin,
    output [3:0] S,
    output Cout
    );
wire [4:0] carry;
genvar i;
assign carry[0] = Cin;
assign Cout = carry[4];
generate
    for (i=0;i<4;i=i+1) begin
        FullAdderLut U (A[i],B[i],carry[i],S[i],carry[i+1]);
    end
endgenerate
endmodule
```





IP Cores

IP Cores

- IP INTEGRATOR
  - Create Block Design
  - Open Block Design
  - Generate Block Design

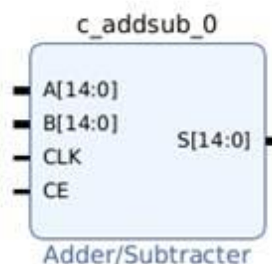
Create Block Design

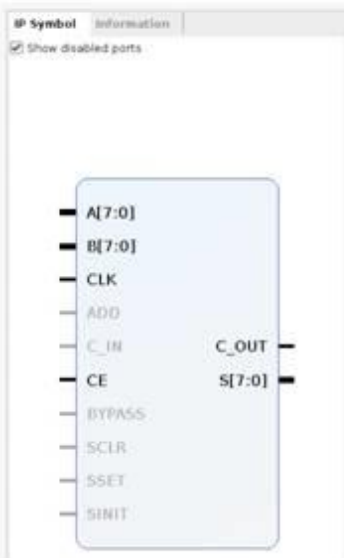
Please specify name of block design.

Design name: Sumador
Directory: <Local to Project>
Specify source set: Design Sources

OK Cancel

This design is empty. Press the + button to add IP.

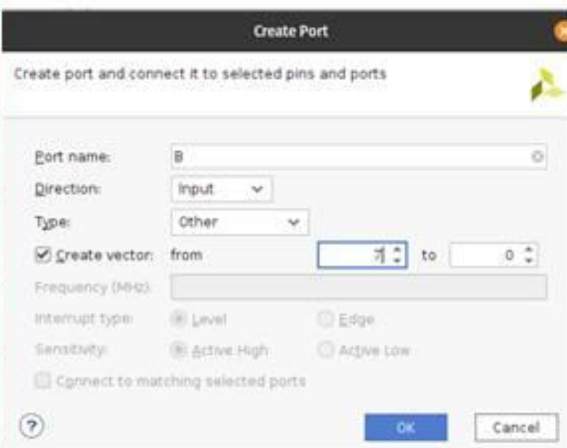


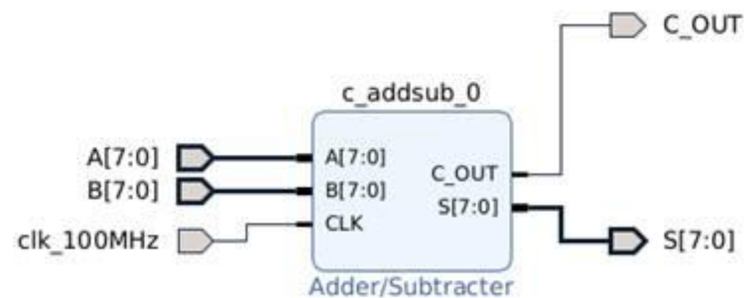


- IP INTEGRATOR
  - Create Block Design
  - Open Block Design
  - Generate Block Design

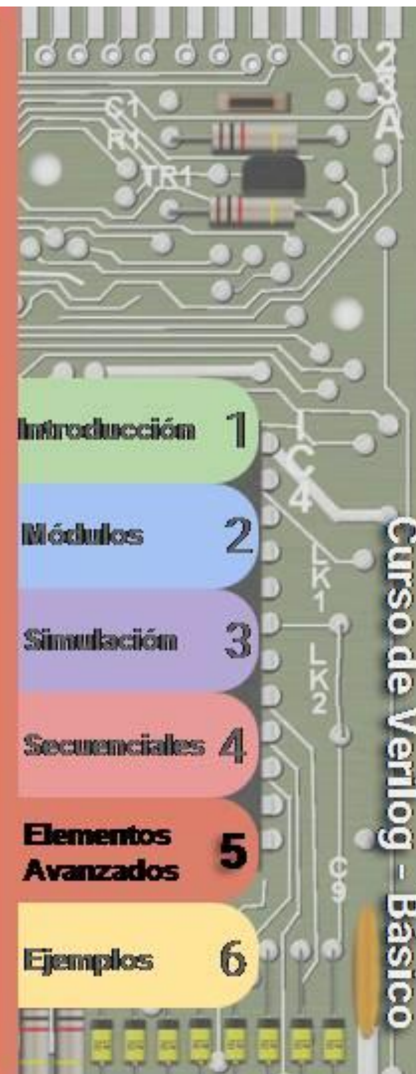
Sumador (Sumador.bd) (1)
Sumador_c_addsub_0_0 (Sumador_c_addsub_0_0.xci)

IP Cores (Create port)





https://www.xilinx.com/support/documentation/ip_documentation/addsub/v12_0/pg120-c-addsub.pdf



IP Cores

Generate Output Products

The following output products will be generated.

Preview

- Sumador.bd (OOC per IP)
  - Synthesis
  - Implementation
  - Simulation
  - Hw_Handoff

Synthesis Options

- Global
- Out of context per IP (selected)
- Out of context per Block Design

Run Settings

- On local host: (selected) Number of jobs: 8
- On remote hosts — Configure Hosts
- Launch runs on Cluster — lsf

Apply | Generate | Cancel

- Sumador_wrapper (Sumador_wrapper.v) (1)
  - Sumador_i : Sumador (Sumador.bd) (1)
    - Sumador (Sumador.v) (1)
      - c_addsub_0 : Sumador_c_addsub_0_0 (Sumador_

Create HDL Wrapper

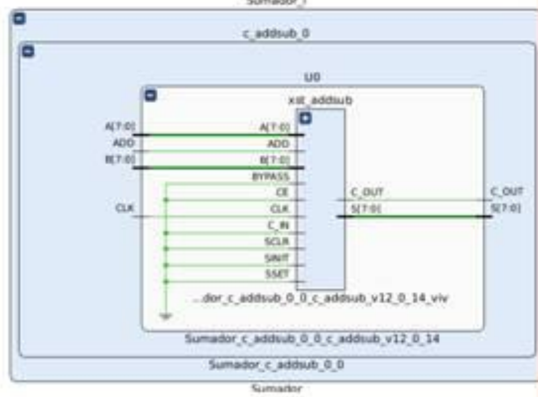
You can either add or copy the HDL wrapper file to the project. Use copy option if you would like to modify this file.

Options

- Copy generated wrapper to allow user edits
- Let Vivado manage wrapper and auto-update (selected)

OK | Cancel

```
module Sumador_wrapper
    (A,
     B,
     C_OUT,
     S,
     clk_100MHz);
  input [7:0]A;
  input [7:0]B;
  output C_OUT;
  output [7:0]S;
  input clk_100MHz;

  wire [7:0]A;
  wire [7:0]B;
  wire C_OUT;
  wire [7:0]S;
  wire clk_100MHz;

  Sumador Sumador_i
       (.A(A),
        .B(B),
        .C_OUT(C_OUT),
        .S(S),
        .clk_100MHz(clk_100MHz));
endmodule
```

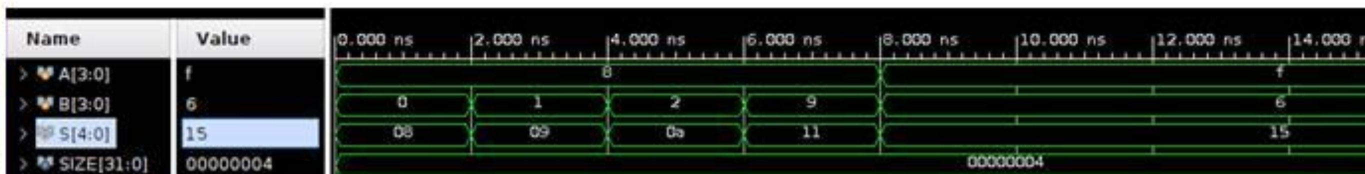


Sumador Simple

Sumador Simple

```
module SumadorSimple
#(parameter SIZE=8)
(
    input [SIZE-1:0] A,
    input [SIZE-1:0] B,
    output[SIZE  :0] S
    );
wire [SIZE:0] Amas,Bmas;
assign Amas = {0,A};
assign Bmas = {0,B};
assign S = Amas + Bmas;
endmodule
```

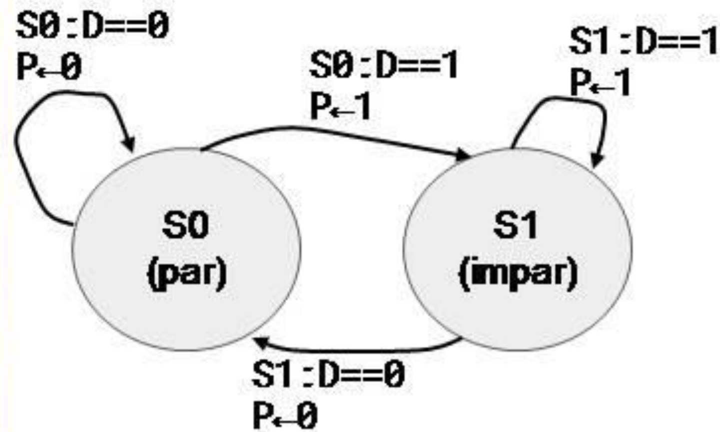
```
module SumadorSimple_tb
#(parameter SIZE=4)
();
reg [SIZE-1:0] A,B;
wire [SIZE :0] S;

SumadorSimple #(.SIZE(SIZE)) U0 (A,B,S);

initial begin
A=4'h8;
B=4'h0;
#2;
A=4'h8;
B=4'h1;
#2;
A=4'h8;
B=4'h2;
#2;
A=4'h8;
B=4'h9;
#2;
A=4'hF;
B=4'h6;
#2;
end
endmodule
```



Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6
Curso de Verilog - Básico

FSM Mealy

FSM (Mealy)

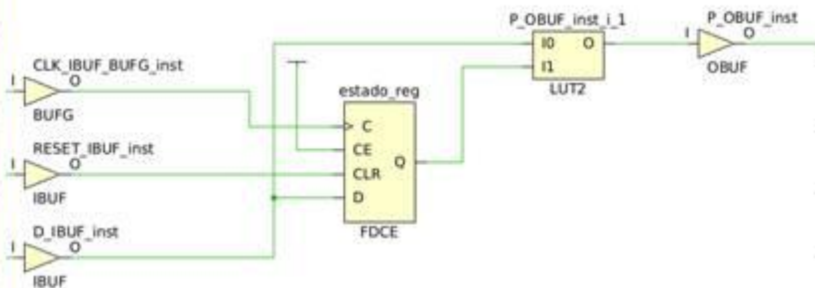


Estado	D	P	Nuevo estado
S0	0	0	S0
S0	1	1	S1
S1	0	0	S0
S1	1	1	S1

FSM (Mealy)

```
module MealyFSM(
    input CLK,
    input RESET,
    input D,
    output reg P
    );
//Definimos valor para guardar el estado
parameter S0=0, S1=1;

reg estado,nuevoEstado;

always @(posedge CLK or posedge RESET)
if (RESET)
    estado <= S0;
else
    estado <= nuevoEstado;
```

```
always @(estado or D)
begin
    P=1'b0;
    case (estado)
        S0: if (D)
            begin
                P=1; nuevoEstado=S1;
            end
            else
                nuevoEstado=S0;
        S1: if (D)
                nuevoEstado=S1;
            else
            begin
                P=0; nuevoEstado=S0;
            end
        default:
            nuevoEstado=S0;
    endcase
end
endmodule
```



Estado	D	P	Nuevo estado
S0	0	0	S0
S0	1	1	S1
S1	0	0	S0
S1	1	1	S1

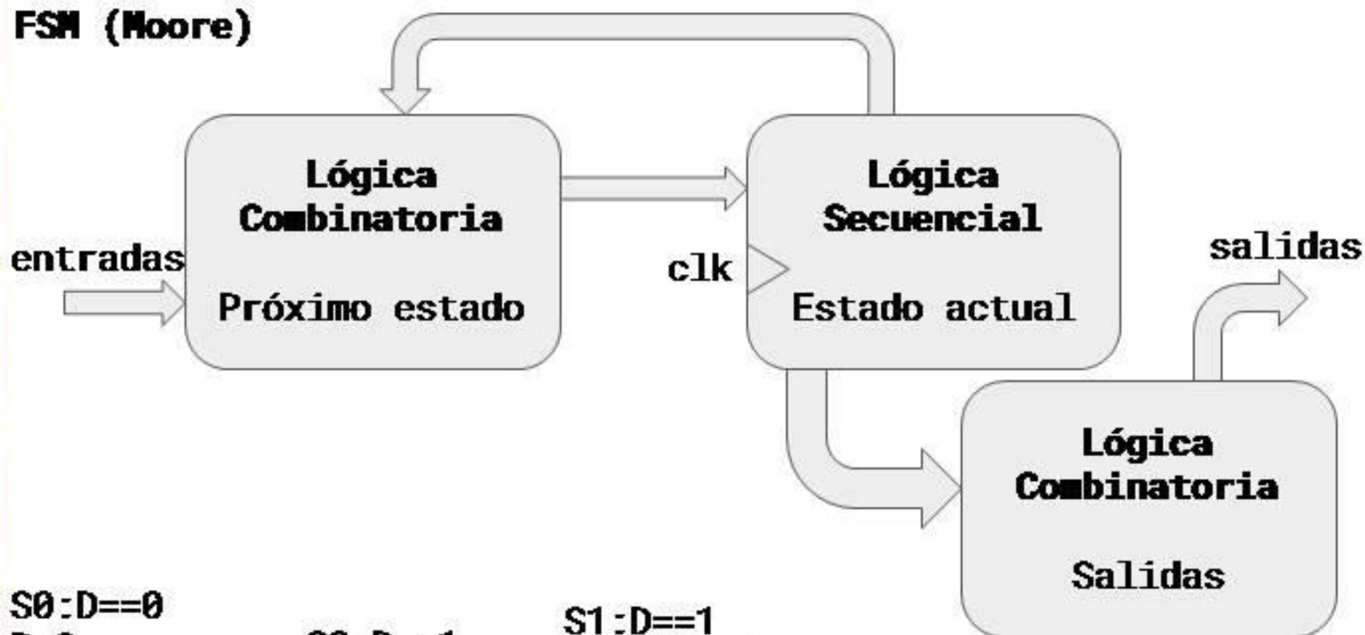
Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6

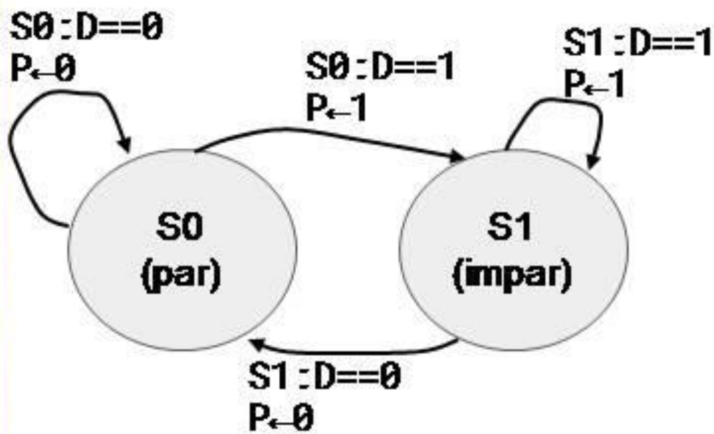
FSM Moore

Introducción 1
Módulos 2
Simulación 3
Secuenciales 4
Elementos Avanzados 5
Ejemplos 6

Curso de Verilog - Básico

FSM (Moore)





Estado	D	P	Nuevo estado
S0	0	0	S0
S0	1	1	S1
S1	0	0	S0
S1	1	1	S1

FSM (Moore)

```
module MooreFSM(
    input CLK,
    input RESET,
    input D,
    output reg P
    );
//Definimos valor para guardar el estado
parameter S0=0, S1=1;

reg estado,nuevoEstado;

always @(posedge CLK or posedge RESET)
if (RESET)
    estado <= S0;
else
    estado <= nuevoEstado;

always @(estado)
begin
    case (estado)
        S0: P=0;
        S1: P=1;
    endcase
end

always @(estado or D)
begin
    nuevoEstado=S0;
    case (estado)
        S0: if(D)
              nuevoEstado=S1;
        S1: if(!D)
              nuevoEstado=S1;
    endcase
end
endmodule
```

Estado	D	P	Nuevo estado
S0	0	0	S0
S0	1	1	S1
S1	0	0	S0
S1	1	1	S1