FPGAs o “Arreglos de Compuertas Programables en Campo”

Son **circuitos integrados digitales que contienen bloques lógicos programables** junto con interconexiones configurables entre dichos bloques (muy grandes , por eso son costosos de producir), el costo de la producción se define según el área de silicio que ocupan. Además, **tienen complejas interconexiones** que lo hacen muy poco eficiente comparado con su área, pero esto lo contrarrestan con su **versatilidad**.

- Un tipo de chip son los **microprocesadores** donde el hardware es fijo. Las funciones se realizan en software.
- Otro formato de chip, **ASIC (Application Specific IC)** : diseñado para implementar una función lógica particular. Son “hechos a medida”. Son más rápidos que las FPGA, consumen menos y, fabricados en gran escala, son más baratos.
- Por último, las **FPGAs** : las funciones se realizan en hardware. No están hechas a medida, por lo que el usuario puede configurarlas de acuerdo a sus necesidades.

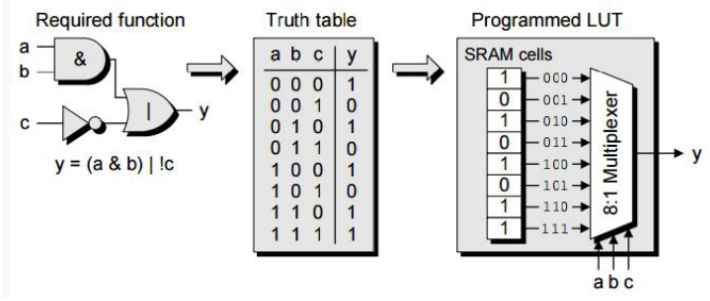
¿Cómo se configuran?

- Algunas FPGA permiten ser programados **una sola vez** (OTP One Time Programmable).
- Otras pueden ser programadas **múltiples veces** “in the field”, es decir, no las programa el fabricante, sino el desarrollador.
- Si un dispositivo puede ser programado **mientras está embebido en un sistema mayor**, se dice que es ISP (In System Programmable).

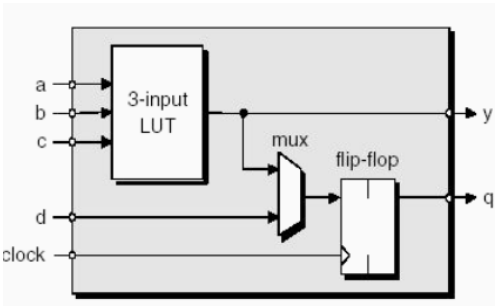
Elementos básicos de las FPGAs y cómo se constituyen (**Elementos lógicos**) :

- Recursos de memoria (pequeños bloques configurables) : al igual que en ODC.
- I/O configurables.
- Recursos de ruteo.
- Recursos adicionales (funciones lógicas que se implementan según convenga).

La **función lógica** se almacena en una **tabla de verdad de 16x1** (para las LUTs de 4 entradas). La columna de **valores de salida** de la función combinacional son los valores que realmente **se almacenan en la LUT**.



- Todas las FPGA se basan en arrays de pequeños elementos de lógica digital. Para usar un determinado dispositivo, los problemas de lógica digital deben ser descompuestos en circuitos lógicos que puedan ser mapeados en una o más de estas “celdas lógicas”.



- Para aplicaciones que requieren acceso a memoria, **las FPGAs cuentan con bloques de memoria disponibles**. La cantidad de bloques disponibles **depende del tamaño** de la FPGA.
- Se pueden conectar diferentes **RAMs en cascada**, ya sea para tener un mayor ancho de la palabra de datos, para tener un mayor tamaño de la memoria o ambos.
- Un mismo bloque de memoria puede ser configurado para que funcione como RAM, ROM, FIFO (First Input First Output), convertidor de ancho de palabra, registro de desplazamientos, etc.

Para poder recibir y transmitir señales digitales, las FPGAs **disponen de un complejo bloque de E/S** que posibilita su uso en muy diversos rangos de tensiones, frecuencias de trabajo, estándares de señales digitales, etc., lo que las hace muy adaptables a las necesidades del sistema del que forman parte. **Existe un bloque E/S por cada terminal de la FPGA**, por lo que **cada una puede ser configurada como entrada, como salida o bidireccional.**

¿Cómo es el flujo de diseño al configurar las FPGAs?
Para ello, se describen los siguientes pasos o procesos.

- **Design Entry** : El circuito deseado es especificado mediante un diagrama esquemático o utilizando algún lenguaje de descripción de Hardware como SystemVerilog.
- **Synthesis** : A partir del diseño ingresado, se infiere la lógica correspondiente y se sintetiza a un circuito utilizando los elementos lógicos (LEs) del chip FPGA.
- **Functional Simulation** : Se verifica la funcionalidad del diseño sintetizado mediante simulación.
- **Fitting** : La herramienta Fitter determina la ubicación de los LEs del diseño en los LEs disponibles en el chip FPGA y elige las interconecciones entre ellos.
- **Configuration** : El circuito diseñado es implementado físicamente en el chip FPGA.

System Verilog

HDL
El lenguaje de descripción de hardware realiza una **caracterización abstracta** del mismo para simular y debuggear una determinada implementación, o diseño.

HDL to Gates
En **simulación**, las entradas se aplican al circuito y las salidas pueden ser chequeadas para corroborar su correcto funcionamiento. Debuggear en simulaciones es muy útil para ahorrarnos tiempo y dinero.

La **síntesis**, transforma el código en HDL en una “netlist” describiendo el hardware, el sintetizador lógico puede generar optimizaciones en orden de reducción del hardware requerido.

SV Modules

Existen dos tipos de módulos, el primero define el comportamiento de ese circuito y lo describe mediante relaciones entre entradas y salidas **(behavioral)**. Mientras que el otro es estructural y describe cómo se construye un módulo a partir de otros **(structural)**.
module/endmodule → sintaxis para iniciar y finalizar la descripción de un módulo.

- Para nombrar un módulo → module “example” →luego listamos las entradas/salidas.
- Las señales lógicas (entradas, salidas) son booleanas y toman el 0 y 1.
- “assign” statement → describe la lógica combinacional

Es importante destacar que ambos tipos de módulos son sintetizados de la misma manera, y sus diferencias se definen por :

Lógicas para la descripción de hardware

Se denomina sistema combinacional o **lógica combinacional** a todo sistema lógico en el que sus salidas son función exclusiva del valor de sus entradas en un momento dado. No intervienen en ningún caso estados anteriores de las entradas o de las salidas, siendo asignaciones concurrentes entre sí, es decir, suceden todas al mismo tiempo. Por ende, las salidas cambian en función de la variación de las entradas.

Por otro lado, **la lógica secuencial** es útil cuando se necesita algún dato o elemento en memoria, y se utiliza dentro de la estructura always.

Estructura Always

El código dentro de este bloque se ejecutará según la lista de sensibilidad descrita dentro del bloque, por ejemplo, always @(lista de sensibilidad). Generalmente esta última, abarca el reset/clock o ambas.

Para representar lógica combinacional con un bloque always :

- La palabra reservada always debe estar seguida de un evento (el símbolo @).
- La lista de sensitividad no debe contener posedge o negedge (palabras que sirven para recibir un flanco).
- La lista de sensitividad debería incluir todos los input del bloque procedural.
- El bloque no puede contener ningún otro control de eventos.
- Todas las variables escritas en el bloque procedural deben ser actualizadas para todos las posibles condiciones de inputs.
- Cualquier variable escrita en el bloque procedural no puede ser escrita por ningún otro bloque.
- **Recomendación :** no usarlo en estos casos. Usar always_comb

Para representar lógica secuencial con un bloque always :

- La palabra reservada always debe estar seguida por un control de eventos sensible a flancos.
- Todas las señales en la lista de sensibilidad deben ser calificadas con posedge o negedge.
- El bloque procedural no puede contener ningún otro control de eventos.
- Cualquier variable escrita en el bloque no puede ser escrita en ningún otro bloque.

Always_comb

- Sirve para inferir lógica combinacional.
- No es necesario especificar una lista de sensitividad, porque como la intención es representar lógica combinacional, se infiere la lista que incluye todas las señales leídas en el bloque procedural.
- Va a saltar una alerta si se usa incorrectamente, por ejemplo si no se define el valor de una variable de salida.
- Se lee de manera secuencial. Por lo tanto se pueden usar estructuras como if ... else y case(z).

Estructura Case

Infiere lógica combinacional solo sí se describen todas las posibles combinaciones de input. Se necesita un caso “default” porque se deben especificar todas las salidas en todos los posibles casos. Un caso particular, es el casez, que permite contemplar condiciones sin cuidado.

Always_ff

Representación de la lógica secuencial, se puede dar el caso de que la salida no esté definida (probablemente haya basura). Se definen señales exclusivas para posedge y negedge.

Nota : El valor anterior va a estar retenido por el flip flop que se está utilizando para implementar este *always*.

Asignaciones dentro de los bloques Always

Se diferencian dos tipos, **bloqueantes** y no **bloqueantes**. El sintetizador inferira de manera distinta dependiendo del caso, es decir, la semántica entre ambas asignaciones es diferente.

- **No bloqueante (<=) :** es concurrente, es decir, ocurre simultáneamente. En general se utiliza para la lógica secuencial.

- **Bloqueante (=) :** utilizada para la lógica combinacional, ocurre en el orden en que aparece en el archivo.

System I/O y Exceptions

Las excepciones son eventos, no branches, que cambian el flujo de la ejecución de las instrucciones en un código determinado.

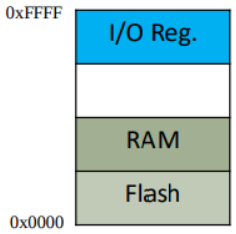
Bus-based I/O

El procesador se comunica con otros dispositivos mediante el uso de determinados pines. Hoy en día, los chipsets utilizan una estructura “Bus-based I/O”, en la cual :

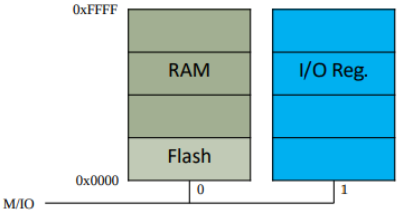
- El procesador tiene puertos de address, data y control que conforman un sistema de bus.
- El protocolo para la comunicación está construido dentro del chip.
- Una sola instrucción realiza el protocolo de lectura o escritura en el bus.

El procesador se comunica con la memoria y periféricos mediante el mismo bus, y aquí, existen dos formas de comunicación :

Memory-mapped I/O : los registros destinados a los periféricos ocupan el mismo espacio direccionable que la memoria, es decir, es el mismo mapa de memoria. Utilizado por arquitecturas AMD - RISK, se accede mediante las instrucciones de Load/Store, y como no existen actualmente demasiadas interfaces I/O, tiene algo más de sentido utilizar esta distribución. Cabe destacar, que se diferencia un rango de direcciones para los puertos y otro para la memoria.

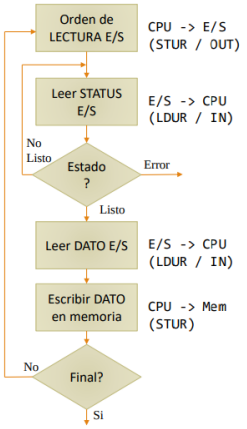


Standard I/O : pines adicionales son utilizados para indicar el acceso a periféricos o a memoria. Utilizado por Intel, se utilizan instrucciones diferentes a las de Assembly para su acceso, para esto, se añade una señal más de control que indica el acceso a memoria o a periféricos. En consecuencia, el mapa de memoria se divide, teniendo como ventaja el evitar pérdida de memoria (espacio direccionable).



Metodos de Operacion - Polling driven (E/S programada)

El manejo se realiza mediante el uso de instrucciones de E/S por código de programa, teniendo como principal desventaja, la pérdida de muchos ciclos de instrucción en revisar el estado del módulo E/S



El polling se realiza por software, incluyendo los lazos de espera, y durante este proceso el microprocesador es inutil. No hay interacción directa entre el procesador y E/S.

En otras palabras, El "polling" es un método de operación utilizado en programación y sistemas de E/S para gestionar la comunicación entre un dispositivo y un programa de computadora. En lugar de utilizar interrupciones o eventos para notificar al programa cuando un dispositivo está listo para interactuar, el "polling" implica que el programa consulte activamente el estado del dispositivo en intervalos regulares para determinar si está listo para realizar una operación de E/S. Características de este método :

- Consulta constante.
- Bucle de polling.
- Eficiencia.
- Simplicidad.

Metodos de Operacion - Interrupciones

En este método, los módulos de E/S pueden interrumpir al procesador. Al finalizar cada ciclo de instrucción, el CPU verifica de manera automática si hay int independientes. De esta manera, el polling no consume ciclos de instrucción ya que es realizado por el CPU mediante HW. Si hay Int, el CPU salta automáticamente a una posición de memoria específica llamada vector de interrupciones, este último, contiene el código (o su referencia) con los procedimientos necesarios para dar servicio a dicha interrupción. El código se denomina ISR (Interrupt Service Routine).

Nota : HW" se refiere a un controlador de entrada/salida. CPU guarda el PC y el status para poder restablecerse.

¿Dónde se aloja la ISR? (interrupt address vector)

Dirección fija (Fixed interrupt) : la dirección está establecida en la lógica del CPU, no puede ser modificada. El CPU puede contener la dirección real, o una instrucción de salto real de la ISR si no hay suficiente espacio reservado.

Dirección vectorizada (Vectored interrupt) : el periférico provee la dirección al CPU por medio del bus de datos. Para esto, se agrega una señal más INT ACK , esta estructura es muy utilizada en sistemas con múltiples periféricos conectados por un bus.

Solución de compromiso : tabla de direcciones (interrupt address table).

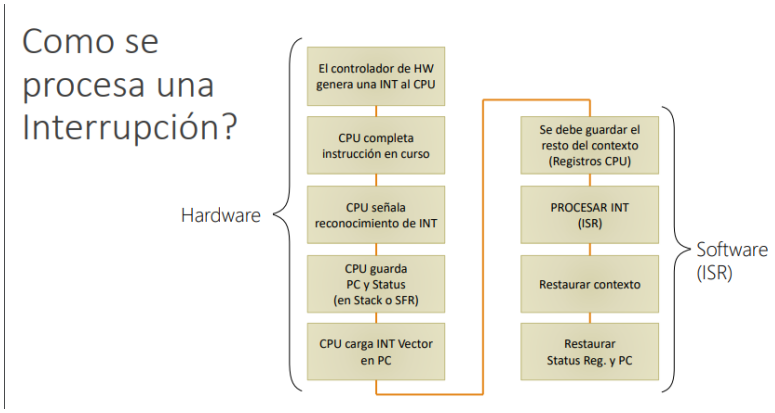


Tabla de direcciones de interrupciones

Es una solución de compromiso entre la interrupción fija y vectorizada. Solo son necesarias dos señales IntReq / IntAck, y se dispone en memoria de una tabla que contiene las direcciones de las ISR. Aquí, los periféricos ya no proveen la dirección de la ISR, sino un índice en la tabla. Las ventajas son menos bits que son enviados desde el periférico, y también, que se puede mover la posición de la IRS sin cambiar los datos de configuración en el periférico.

Interrupciones Enmascarables vs. no-enmascarables

Enmascarables : El programador puede modificar un bit que causa que el procesador ignore una solicitud de interrupción (GEI). Muy importante para usar cuando se tienen porciones de códigos temporalmente críticos.

No-enmascarable : una señal separada que no puede ser ignorada (internas a la CPU). Típicamente reservada para situaciones drásticas, como la ocurrencia de un fallo de alimentación, un acceso indebido a memoria flash o la detección de un código de operación invalido. Estas ISR se las suele conocer con el nombre de “Traps”.

Salto a una ISR

- Algunos procesadores tratan el salto de una INT como una llamada a cualquier otra rutina.
- Salvan el entorno completo (PC, status, registers) – toma muchos ciclos.
- Otros salvan parcialmente su entorno (solo PC y status).
- El programador debe asegurarse que el ISR no altere los registros o debe salvarlos previamente.

- Es necesario instrucciones de retornos diferentes en su set de instrucciones (RET – RETI).

Usamos el término **excepción** para referirnos a cualquier cambio de control si es no-enmascarable y en general es interna; usamos el término interrupción sólo cuando el evento fue causado externamente, estos son enmascarables.

Múltiples periféricos : Arbitraje

Considere la situación donde muchos periféricos solicitan el servicio de una CPU simultáneamente (microcontrolador) . ¿Cuál será atendida primero y en qué orden?

Software polling :

- Muy simple implementación por HW (una sola línea de INT).
- Se va preguntando a cada periférico si fue el generador de la interrupción.
- El programador debe determinar en la ISR el origen de la INT buscando banderas INT FLG.
- La prioridad es establecida en la ISR según el orden de la búsqueda.

Árbitro de prioridades (Priority arbiter) :

- Un módulo que se encarga de gestionar pedidos de interrupciones. Y este es quien interrumpe al procesador.
- Es el más utilizado.

Conexión en cadena (Daisy chain) :

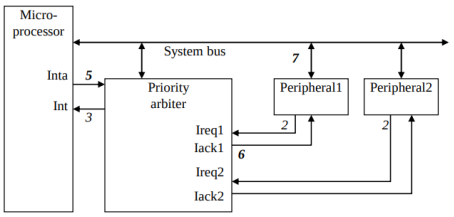
- Barato de implementar pero poco versátil.
- La prioridad está dada por la posición de los periféricos.
- La interrupción se va propagando hasta llegar al micro.

Arbitraje de bus (Network-oriented)

- Utilizado en arquitecturas de múltiples procesadores.
- El periférico debe primero obtener la sesión del bus para luego requerir una interrupción.

El **árbitro de prioridades**, también conocido como **controlador de interrupciones**, es muy usado en arquitecturas de una CPU con propósito general. El periférico hace una petición “INT REQ” al controlador, y este a la CPU. En orden inverso, con los “INT ACK”. Se contempla un esquema de direccionamiento vectorizado, y se definen múltiples esquemas de prioridad :

- Fija
- Round Robin
- FIFO



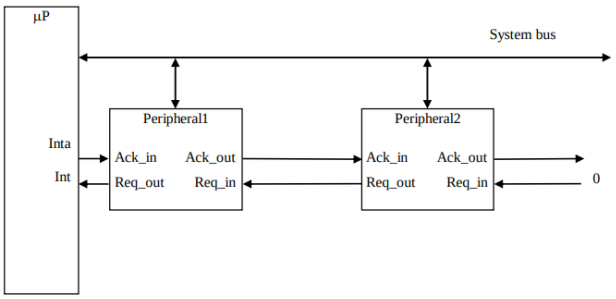
1. CPU ejecuta su programa principal
2. P1 solicita servicio seteando *Ireq1*. P2 también solicita servicio mediante *Ireq2*.
3. Como el controlador detecta al menos una *Ireq* , entonces setea *Int*.
4. La CPU para la ejecución del programa y guarda el contexto.
5. La CPU setea *Inta*.
6. El controlador setea *Iack1* para responder a P1.
7. P1 pone la dirección del vector en el bus de datos
8. La CPU salta a la dirección del ISR leída del bus, la ISR se ejecuta y retorna.
9. La CPU resume la ejecución del programa principal

Conexión en cadena

La lógica de control de arbitraje está embebida en cada periférico, se agrega una señal de entrada y de salida respectivamente, “IntReq” e “IntAck”. Los periféricos están dispuestos en el sistema en forma de cadena, donde un solo periférico se conecta al CPU (el de mayor prioridad), los otros “aguas arriba”. Estos difunden la señal “IntReq” hacia el CPU y la señal “IntAck” hacia el origen de la interrupción.

Pros vs. contras

- Ideal si es necesario agregar o sacar periféricos.
- Bajo rendimiento con muchos periféricos.
- Es necesario agregar lógica por seguridad (periférico fuera de servicio).
- Esquema de prioridad único.



Excepciones e interrupciones

Eventos inesperados que requieren un cambio en el flujo de control, las distintas ISAs se refieren a estos términos con conceptos diferentes. Las excepciones pueden provenir del CPU (opcode invalido, overflow, syscall), mientras que las interrupciones son enviadas por

controladores E/S externos. Lidiar con ellos sin sacrificar performance es una tarea muy difícil.

Manejo de excepciones

Es importante que el procesador, ante estos eventos, pueda :

- Guardar el PC de la instrucción interrumpida : en LEGV8 se guarda en el registro ELR (Exception Link Register).
- Guardar el indicador del problema : se realiza en el registro ESR (Exception Syndrome Register). Asumimos que este indicador es de 1 bit (0 por opcode indefinido y 1 por overflow)
- Guardar el valor que hubiese tenido la dirección de la próxima instrucción que se hubiera ejecutado : ERR (Exception Return Register) Es propio de nuestro procesador, sirve para saltos, donde la instrucción a la cual quiero volver no es el valor de PC + 4. Saca el valor de lo que se hubiera cargado en el PC de no ser por la excepción.

Procesador con Excepciones

Se agregan las interfaces ExtIRQ (External Interrupt Request) y ExtAck (External Interrupt acknowledge) para que el procesador pueda atender excepciones. Ya sean señales producidas por un periférico o el árbitro de prioridades.

Además, se agregan dos nuevas instrucciones que servirán para abordar estos eventos inesperados.

ERET (Exception Return) : es de tipo R, se utiliza para retornar desde una rutina de manejo de excepciones al flujo de ejecución normal del programa. Se utiliza para limpiar y restaurar el estado de la CPU después de manejar una excepción y permitir que la ejecución del programa continúe desde el punto en que se interrumpió.

Antes de ejecutar "ERET," la rutina de manejo de excepciones generalmente debe haber guardado el estado del procesador, incluyendo registros de propósito general, el contador de programa (PC) y otros registros relevantes. "ERET" se encarga de restaurar este estado almacenado y permitir que la ejecución continúe desde donde se interrumpió.

MRS (Move from SystemReg to GeneralPurposeReg) : es de tipo S (nuevo), esta instrucción se utiliza para leer el valor de ciertos registros del sistema en el procesador (ERR, ELR, ESR) y los mueve a los registros de propósito general. En otras palabras, se utiliza para copiar el contenido de un registro del sistema en un registro general propósito visible para el programador.

- **Sintaxis y ejemplo :** MRS R0, CPSR →Lee el estado actual de CPSR en R0.

Algunas instrucciones útiles, implementadas en el procesador :

- Br → Branch to Register.
- SVC →Genera una excepción con 16 bits.

- MSR →Copia la información de un registro de propósito general a un SystemReg.

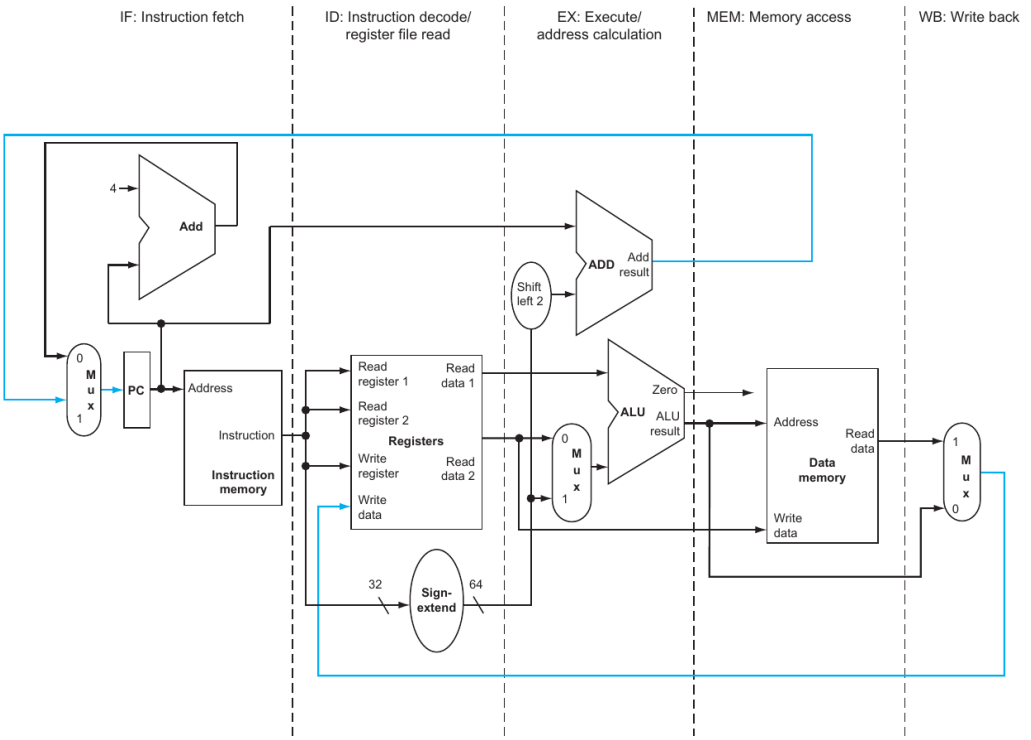
Procesador con Pipeline

El pipelining es una técnica de implementación en la cual se superponen múltiples instrucciones durante su ejecución. De esta manera, se optimiza el microprocesador y no quedan recursos sin uso durante un cierto tiempo.

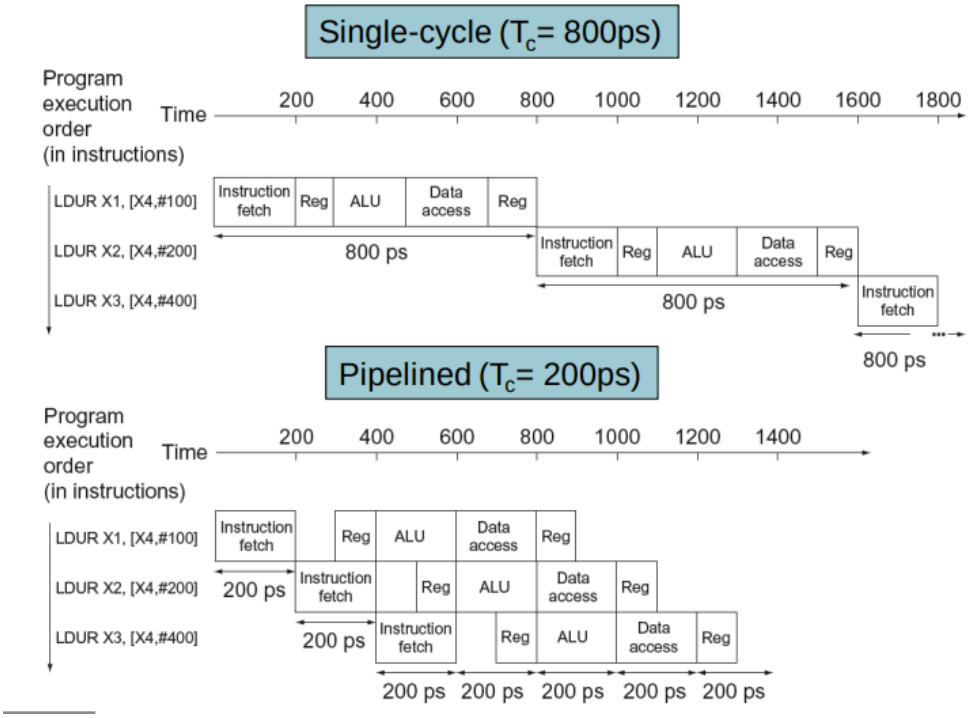
La idea es básica, dividir el proceso de ejecución de las instrucciones, en etapas. Se definen 5 de estas :

- IF → Instruction Fetch.
- ID → Instruction Decode.
- EX → Execute Operation .
- MEM → Access memory.
- WB → Writes the result into the Register.

Generalmente, la técnica de pipeline mejora el rendimiento hasta 5 veces.



Ejemplo de ejecución de instrucciones con un procesador con la técnica de Pipeline :



Asumamos que todas las etapas duran lo mismo, luego se cumple que :

- T_pipelined = T_nonpilelined / num_de_etapas.
- T_pipelined = "Tiempo entre instrucciones con pipeline".
- T_nonpipelined = "Tiempo entre instrucciones sin pipeline".

Dependencia de datos

Caracteriza el hecho de que una instrucción genera un dato o lo modifique, y que otra accede a los mismos para realizar ciertas acciones.

Por ejemplo, la siguiente secuencia de instrucciones tiene una dependencia en X10 :

Add X10, X9, X8

Sub X1, X10, X2

La dependencia de datos es propia de nuestro código, y los posibles “riesgos” o Hazards provienen del hardware.

Hazards

Un hazard es una situación en Pipelining en la cual no se puede ejecutar la siguiente instrucción en el siguiente ciclo de clock.El hazard ocurre cuando una dependencia de datos causa problemas. Y esto ocurre cuando un dato se necesita antes de que se genere (por que se genera en las últimas etapas y se necesita en las primeras de las siguientes instrucciones. Hay muchos tipos de dependencia de datos. A medida que se complejizan las arquitectura van a aparecer más data hazards. Las que vemos nosotros ahora son RAW (read after write).

Hazards Estructurales

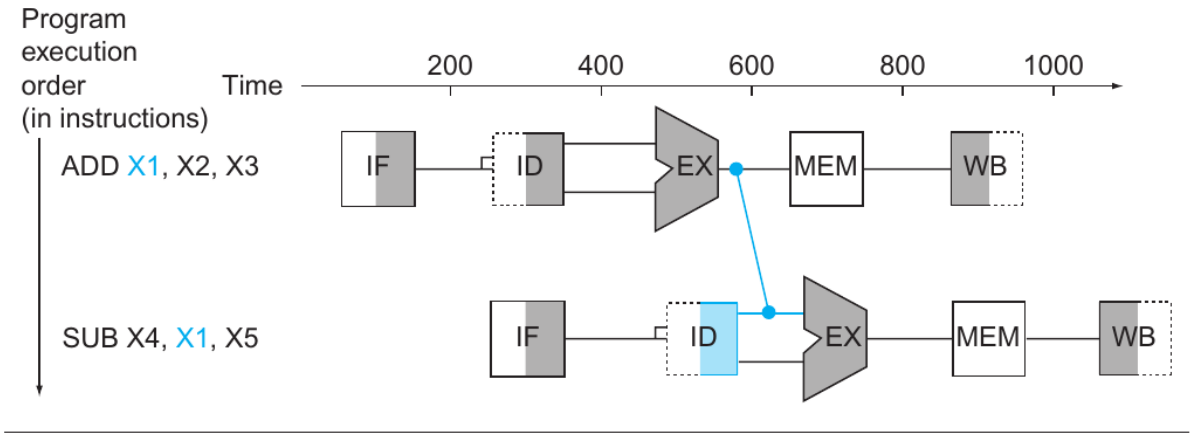
Cuando una instrucción planeada no puede ejecutarse en el ciclo de clock correcto porque el hardware no soporta esa combinación de instrucciones que deben ejecutar. Dos instrucciones necesitan al mismo tiempo el mismo recurso. Se soluciona separando las memorias de datos y la de instrucciones (si el hazard se ocasiona por la memoria) o esperar un ciclo.

Hazards de datos

Cuando una instrucción planeada no puede ejecutarse en el ciclo de clock correcto porque los datos que se necesitan para ejecutar la instrucción todavía no están disponibles. Una instrucción depende de que se complete el acceso a datos en una instrucción anterior. Para solucionarlo se hace un forwarding, stall o forwarding-stall.

Forwarding

Es un método que consiste en retribuir el dato faltante por buffers internos en vez de esperar a que lleguen a través de registros o memoria visibles para el programador. Se usa el resultado ni bien se computa (EX), no se espera a que se guarde en un registro. Requiere conexiones extra en el datapath. Un dato está listo en la etapa de ejecución, no hace falta esperar hasta que se escriba en el registro para usarlo. Se agrega un cable que lleve ese dato ni bien esté listo. Entonces una instrucción de tipo R ya no tiene problemas de dependencia. El forwarding se realiza de Alu-Alu o de Mem-Alu.



Forwarding stall

Los caminos de forwarding **solo son válidos** si la etapa de destino está después en tiempo que la etapa de origen. Frenar el flujo de instrucciones hasta que una esté lista. Un **stall** o burbuja es una pausa iniciada para resolver un hazard.

Hazard de Control

Cuando la instrucción planeada no se puede ejecutar en el ciclo de reloj correcto porque la instrucción que fue fechada no es la que se necesita, es decir, el flujo de direcciones de instrucciones no es el que la pipeline esperaba. Si una instrucción es un salto, recién sabe si tiene que saltar luego del decode. Entonces, en ese tiempo, se "meten" instrucciones de más. Se debe hacer un flush para limpiar esas instrucciones y que no generen cambios visibles.

Hay dos soluciones:

- Hacer forwarding y esperar para saber si hay que saltar o no. Ni bien se sabe que la instrucción se trata de un branch, esperar hasta que la pipeline determine el resultado del branch y sepa qué instrucción fetcher.
- Predecir el branch, se asume un resultado y proceder a partir de esa suposición, en vez de esperar al resultado de verdad. Esta opción no desacelera el Pipelining cuando se acierta.

Predicción de saltos

Existen dos técnicas de predicción, estática o dinámica. Las predicciones estáticas asumen que el salto se va a realizar (taken) o, que no se va a realizar (not taken). En caso de no acertar, la penalización es de 3 ciclos, ya que el micro se da cuenta de que no acertó en su decisión en la etapa de Mem.

- Not taken : Por defecto se asume que el salto no se va a tomar y se carga la siguiente instrucción. En caso que el salto deba tomarse, se hace flush y se realiza el fetch de la instrucción correcta.
- Taken : Por defecto el procesador asume que el salto se toma y carga la instrucción que indica el salto.

Mientras que las predicciones dinámicas, asumen que el salto se realiza o no dependiendo del historial de saltos ejecutados.

Implementación del microprocesador con Pipeline

Se agregan registros de pipeline para mantener datos así las porciones del path de una instrucción puede ser compartida durante la ejecución de la misma. Los registros deben ser suficientemente grandes para guardar toda la información correspondiente a las líneas que los atraviesan. Para pasar algo de una etapa temprana del pipeline a una más tardía, la información debe ser guardada en un registro de pipeline, de otra manera, la información se pierde cuando la siguiente instrucción entra a esa etapa del pipeline.

Pipelined datapath