Documentación oficial

Procesador segmentado

Arquitectura de computadores

Equipo:

Sergio Anguita

Aitor Brazaola

Rubén García

Profesor:

Jose Luis Gutierrez Temiño

24 de noviembre de 2014

Índice

Introducción

El siguiente documento compone la documentación total del proyecto opcional de Arquitectura de computadores, asignatura del tercer curso de la Universidad de Deusto, en el cual se dio la oportunidad de crear un procesador MIPS segmentado siendo lo mas fiel posible a los estándares. Como objetivo, crear y diseñar desde cero un procesador segmentado totalmente funcional que entienda el set de instrucciones dado por el profesor. Del mismo modo, el procesador tiene que ser capaz de detectar los conflictos que surjan durante la ejecución de cualquier programa y controlarlos. Este procesador, está basado en el MIPS, solo que con un set de instrucciones mucho menor, con un total de nueve instrucciones. El procesador se divide en las siguientes zonas:

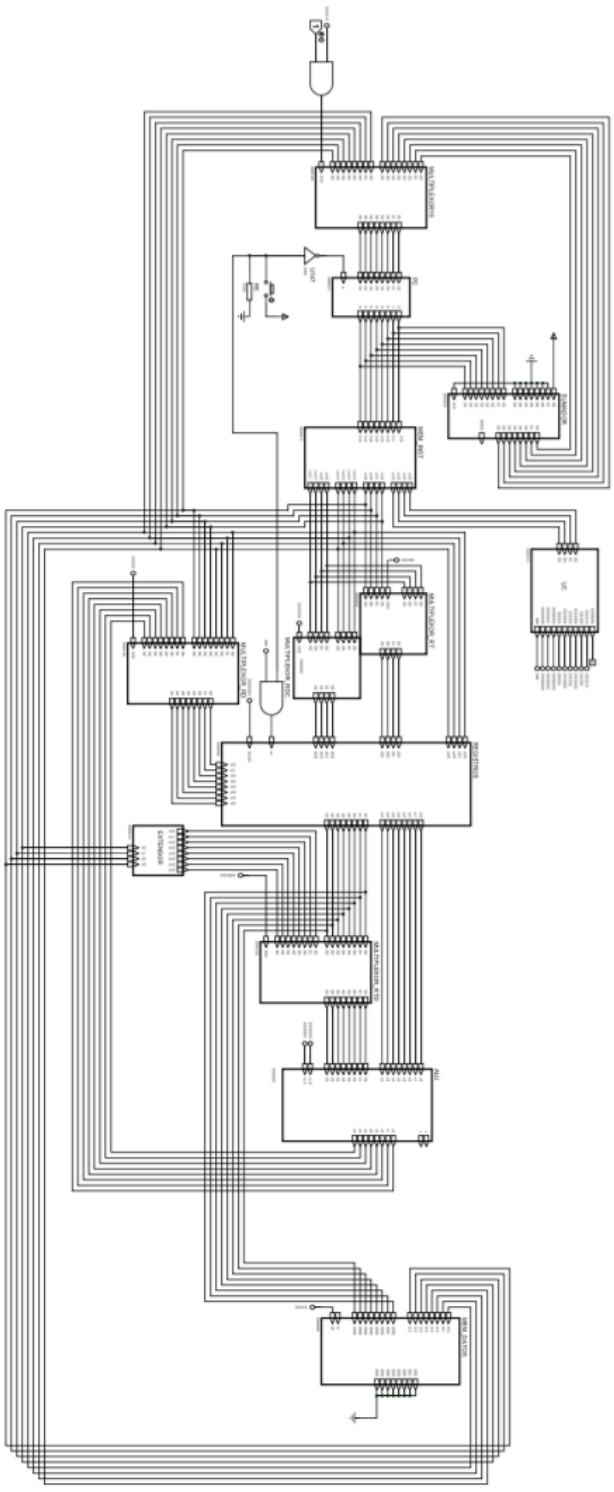
- **Registro PC:** Puede ser por incremento de uno, para avanzar. o por salto. Estas dos son las únicas formas que dispone el contador de programa para avanzar a la siguiente instrucción. Este registro almacena la instrucción y en el cambio de ciclo pasa a apuntar a la siguiente instrucción a ejecutar. Dispone de una señal que le permite decidir si tiene que ejecutar un salto de instrucción o seguir de manera secuencial. Esta decisión se realiza introduciendo ambas direcciones por un multiplexor, y en función de la señal de control de ese multiplexor, se elegirá una u otra.
- **Memoria de instrucciones:** Apuntada por el PC saca la dirección de 16 bits, con su código de operación como identificativo de dicha instrucción y luego mediante una serie de multiplexores se decide el camino de datos que la instrucción debe seguir para completarse correctamente.
- **Registros:** Dispone de dos bancos de registros: un banco *rs* y otro banco *rt*. Los valores a escribir en el banco de registro se introducen por *rd* y *rdd*. Para guardar un dato en los registros, aparte de el dato a guardar, se necesita especificar en qué registro se va a guardar. Este dato de entrada se especifica mediante los valores *rs* y *rt* que actúan como punteros del banco de registros. Todas estas operaciones están sincronizadas por el reloj del sistema.

- **Unidad de Control:** Es la encargada generar las señales de control a partir del código de operación de cada instrucción.
- **Unidad lógica-aritmética:** Es la encargada de realizar las operaciones lógico aritméticas del procesador a partir de dos datos de entrada. En este caso, las únicas operaciones que la ALU realiza son: la operación lógica AND, la operación lógica OR y desplazamiento de un bit (instrucción LSL).
- **Unidad de detección de conflictos:** Es la encargada de detectar y controlar los posibles conflictos que puedan surgir mientras se ejecutan varias instrucciones en distintas etapas. Recibe como entrada el código de operación, la mayoría de señales de control y las señales de control de los registros de cada etapa en las que puede haber conflictos.
- **Memoria de datos:** Se almacenan y se leen los datos de las instrucciones que lo soliciten.Todo en conjunto, se controla mediante la unidad de control que decide si almacenar datos o leerlos, y qué multiplexores activar para lograr el correcto funcionamiento.

Existen también elementos secundarios como un extensor de 4 bits a 8 bits, un sumador que permite sumar uno al contador de programa, registros interetapa, y diversas puertas lógicas utilizadas tanto en el interior de la ALU como fuera de ella, así como en los lugares donde se han hecho necesarias: como en el cambio de ciclo del registro. Por último los multiplexores son un elemento secundario pero de gran importancia que siempre hay que tener en cuenta, así como sus señales de control.

Diseño

Se partirá del procesador MIPS sin segmentar creado en las practicas de la asignatura al cual se le añadirán, entre otros elementos, cinco etapas usando cuatro registros interetapa y una unidad de control de errores.



El set de instrucciones

Como el set de instrucciones esta compuesto por nueve (originalmente ocho) instrucciones y cada instrucción es de 16 bits , es necesario dedicar al menos 4 bits (ya que con $2^4=16$ *Instrucciones* tenemos 16 instrucciones disponibles) al campo *'Código de operación'*. Por lo tanto, tendremos 12 bits para definir el resto de campos de la instrucción. Una vez definidos todos los campos, esta es la distribución inicial de campos elegida para todas las operaciones:

Set de instrucciones

MOVIS	0	0	0	0	d	a	t	o					r	d	s	
MOVIT	0	0	0	1	d	a	t	o					r	d	t	
JRZ	0	0	1	0	d	i	r									
CLC	0	0	1	1												
LSL	0	1	0	0					r	s						
MOVMT	0	1	0	1	d	i	r						r	t		
ORI	0	1	1	0	d	a	t	o	r	s			r	d	s	
ANDS	0	1	1	1	r	t			r	s			r	d	s	

A este set de instrucciones se le añadió otra instrucción más para añadirle un poco más de dificultad al procesador y así tener que controlar conflictos por dependencia de datos entre las distintas instrucciones que escribían y leían de la memoria de datos. Al añadir otra instrucción más, la distribución del set de instrucciones se modifica quedando de la siguiente forma:

Set de instrucciones

MOVIS	1	0	0	0	d	a	t	o					r	d	s	
MOVIT	0	0	0	1	d	a	t	o					r	d	t	
JRZ	0	0	1	0	d	i	r									
CLC	0	0	1	1												
LSL	0	1	0	0					r	s	d	s				
MOVMT	0	1	0	1	d	i	r						r	t		
ORI	0	1	1	0	d	a	t	o	r	s			r	d	s	
ANDS	0	1	1	1	r	t			r	s			r	d	s	
MOVMRT	1	0	0	1	d	i	r						r	d	t	

Del mismo modo, también se ha creado un juego de instrucciones de prueba para probar el procesador una vez que el procesador esté terminado. Las instrucciones tienen un orden determinado el cual se presenta a continuación y se explicará detalladamente en el apartado *Instrucciones* de este documento. Cada instrucción realiza un tipo de operación diferente y para cada instrucción se generan las señales de control adecuadas para los elementos del procesador.

Ejecución de instrucciones

1. Instrucción MOVIS

1.1. Descripción

Coge el dato, lo lleva a través del Multiplexor RD con un 0 y lo guarda en el registro RS.

1.2. Valores guardados (hexadecimal)

Código	Dato parte 1	Dato parte 2	RDS
0	2	5	0

Guardamos así el dato en hexadecimal el valor 2 y 5 o 0010 0101 en el registro 0.

2. Instrucción MOVIT

2.1. Descripción

Coge el dato, lo lleva a través del Multiplexor RD con un 0 y lo guarda en el registro RT. Además de el Multiplexor RDC con un 1.

2.2. Valores guardados (hexadecimal)

Código	Dato parte 1	Dato parte 2	RDT
1	2	8	1

Guardamos así el dato en hexadecimal el valor 2 y 8 o 0010 1000 en el registro 1.

3. Instrucción LSL

3.1. Descripción

Coge el dato de RS, lo desplaza a la derecha y lo vuelve a guardar en rs en la misma dirección activando el multiplexor RD. El multiplexor RDC se pone a 0.

3.2. Valores guardados (hexadecimal)

Código		RS/RDS	
4		0	

En la dirección indicada de RS, sacamos el valor que anteriormente habíamos metido en la dirección 0, en hexadecimal 2 y 5 o 0010 0101. Lo desplazamos un bit dando 0100 1010 y lo almacena en la dirección 0.

4. Instrucción MOVMT

4.1. Descripción

Con la dirección pasada como 8 bits indicamos esta dirección en memoria, sacar el valor de RT, activando el Multiplexor RDC y meterlo en memoria el dato extraído.

4.2. Valores guardados (hexadecimal)

Código	DIR Parte 1	DIR Parte 2	RT
5	2	3	1

En la dirección indicada con el hexadecimal 2 3 o 0010 0011 almacenar en memoria el valor de RT que hemos guardado con MOVIT en 1 sacando así el valor 0010 1000*

5. Instrucción ORI

5.1. Descripción

Con el dato, se extiende, luego se coge para pasarlo a ALU con el Multiplexor RTD y se pasa rs también. Se realiza la instrucción OR y se pasa de nuevo y con el Multiplexor RD se almacena.

5.2. Valores guardados (hexadecimal)

Código	Dato	RS	RDS
6	D	0	4

El dato será D, es 1101, pero será extendido al dato 1111 1101 para poder compararse con el dato que proviene de rs, que es el dato anteriormente extendido siendo de 0100 1010. Esta vez, para variar almacenaremos el dato en otro registro, el 4, o 0100.

6. Instrucción ANDs

6.1. Descripción

Consiste en sacar los valores de RS, RT, hacer la operación AND en la ALU y guardarlo de nuevo en los registros. Para esto tenemos que pasar un 0 por el multiplexor llamado RTD, luego por el multiplexor RT un 0 y por el de RDC un 1. Por último el de RD.

6.2. Valores guardados (hexadecimal)

Código	RT	RS	RDS
7	1	4	4

Sacando el primer valor que guardamos en RT 1 que es 0010 0011 y el valor de RS en 4 que es el resultado de la operación anterior, hacemos el cálculo de la AND y lo almacenamos nuevamente en el registro 4.

7. Instrucción CLC

7.1. Descripción

Pone a 1 un bit de ClearC que sale de la ALU que lo que haría sería poner a 0 el bit C. Al no utilizarse dicha salida no se ha llevado hasta el.

7.2. Valores guardados (hexadecimal)

Código			
8			

8. Instrucción JRZ

8.1. Descripción

Salta a la dirección especificada en el campo de 8 bits dir para lo cual tenemos SELDIR que activa el Multiplexor10 y además el usuario puede decidir con un bit que puede especificar si realmente quiere saltar o continuar avanzando en memoria.

8.2. Valores guardados (hexadecimal)

Código	**DIR Parte 1**	**DIR Parte 2**	
8	0	0	

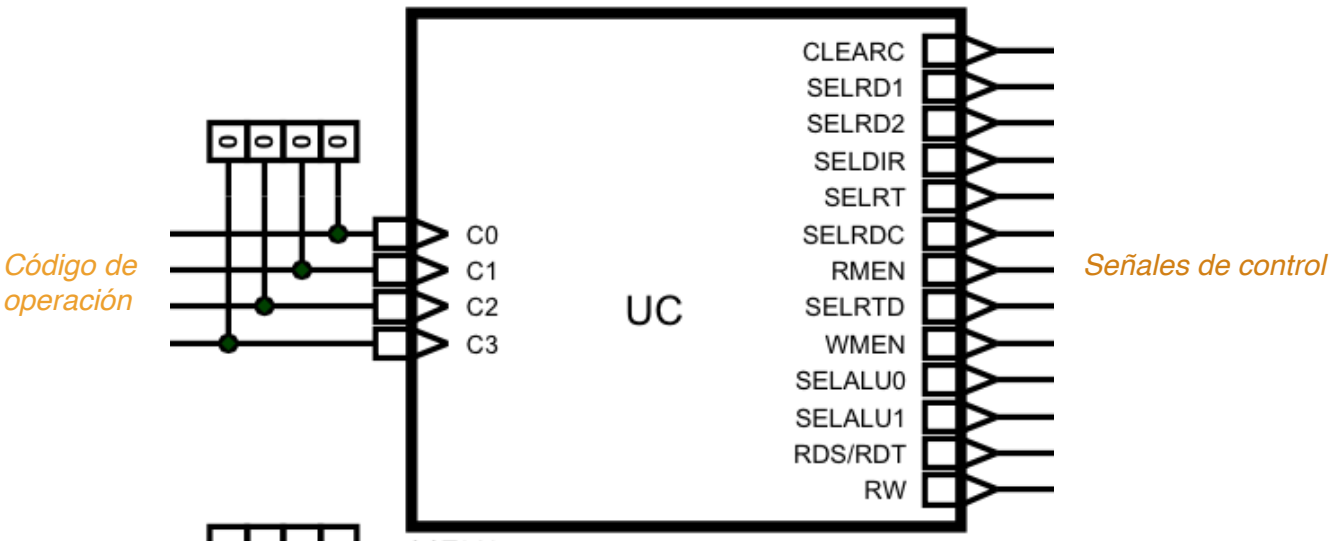
Indicamos el valor 0 para reiniciar la secuencia de instrucciones insertada.

Unidad de control

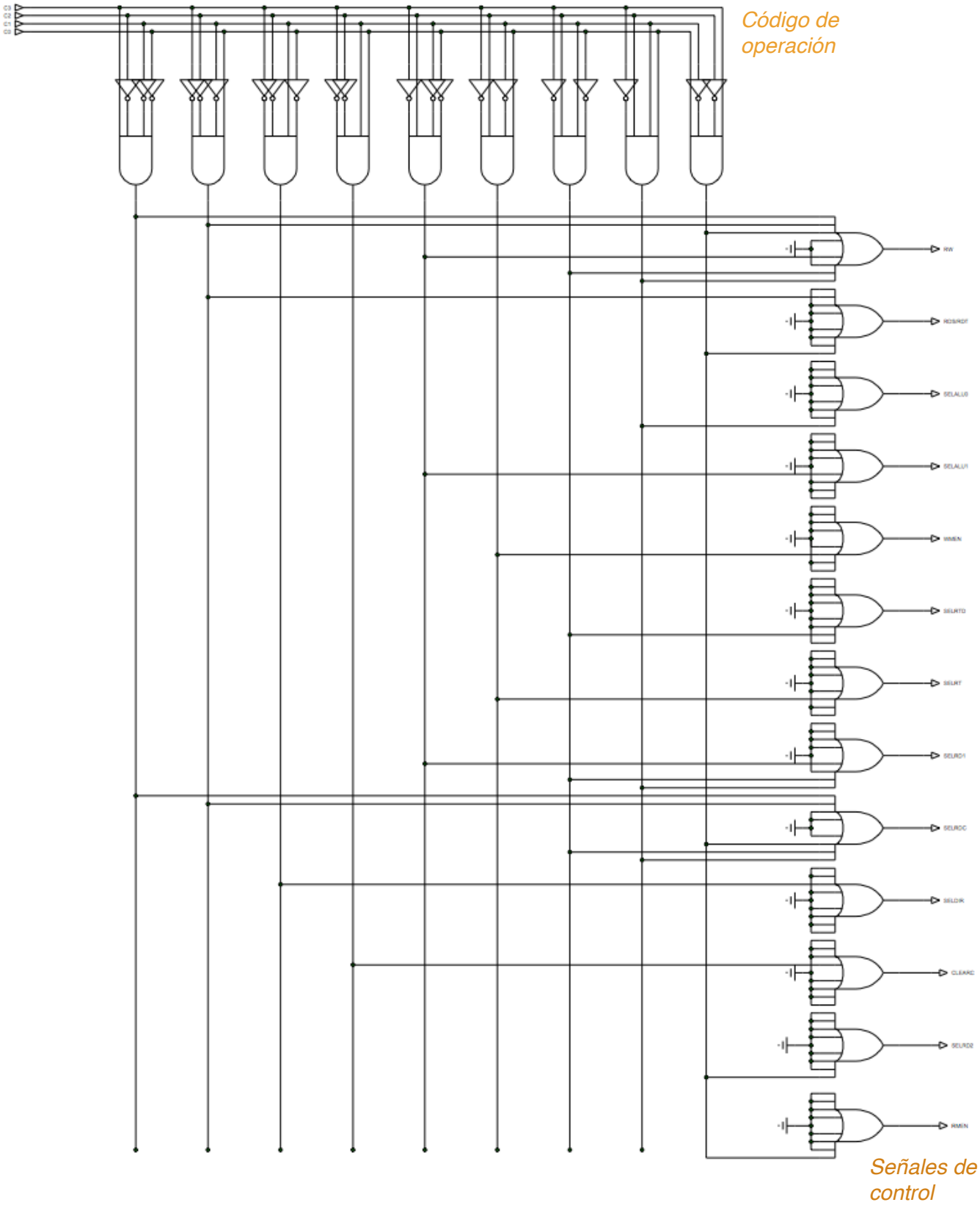
La unidad de control es la encargada de generar las señales de control del procesador a partir del código de operación de cada instrucción. Para diseñar la unidad de control, es necesario hacer el diseño de cada instrucción en una tabla en la que se asignaran los valores 1 o 0 a cada valor de salida a controlar. La tabla debe contemplar todas las señales de control y todas las instrucciones. A continuación, se muestra la tabla de de control de señales para nuestro set de instrucciones.

Unidad De Control

	RW	RDS/RDT	SELALU	WMEN	SELRTD	SELRT	SELRD0	SELRD1	SELRDC	SELDIR	CLEARC	RMEN
MOVIS	1	0	XX	0	X	X	0	0	1	0	0	0
MOVIT	1	1	XX	0	X	X	0	0	1	0	0	0
JRZ	0	X	XX	0	X	X	X	0	X	1	0	0
CLC	0	X	XX	0	X	X	X	0	X	0	1	0
LSL	1	0	10	0	X	X	1	0	0	0	0	0
MOVMT	0	X	XX	1	0	1	X	0	X	0	0	0
ORI	1	0	0	0	1	X	1	0	1	0	0	0
ANDS	1	0	1	0	0	0	1	0	1	0	0	0
MOVMRT	1	1	0	0	0	0	0	1	1	0	0	1



Diseño de la unidad de control en Proteus



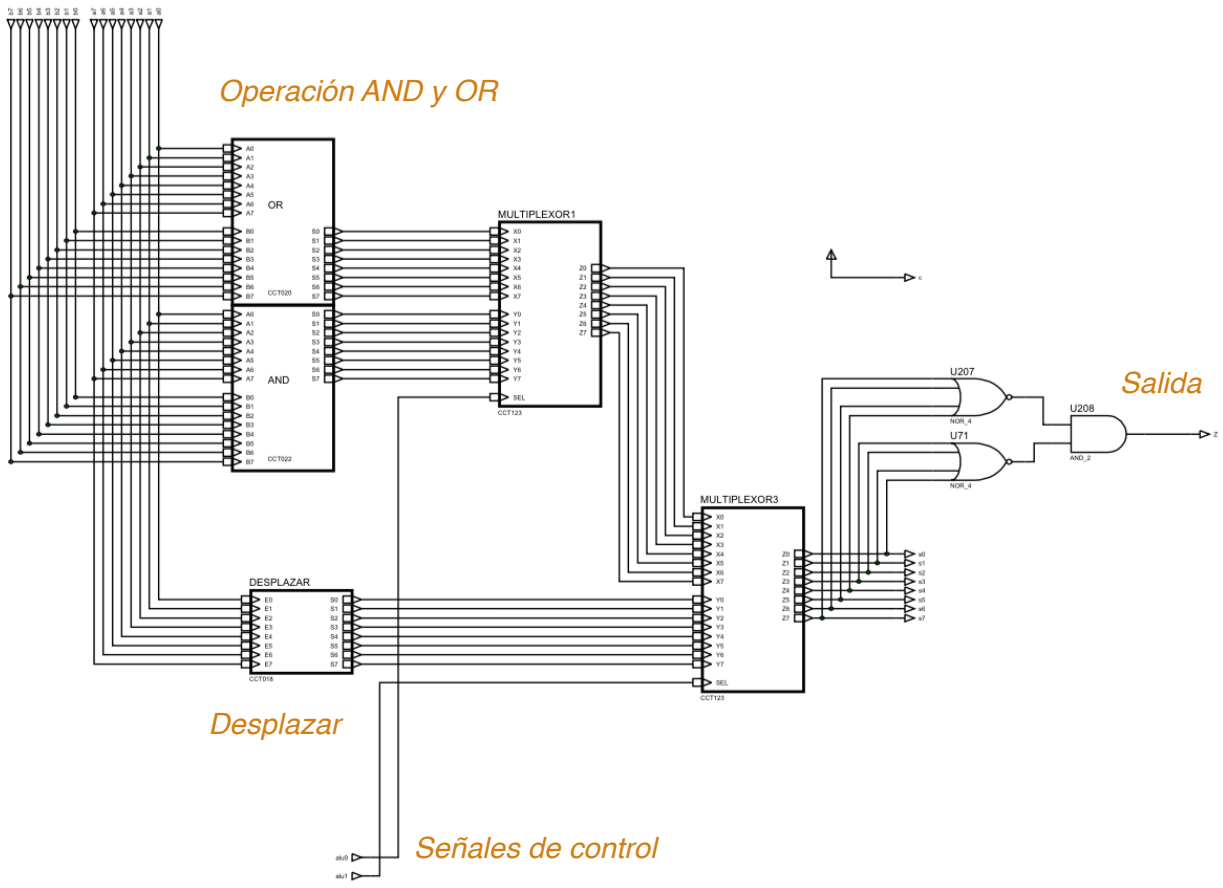
Diseño de la unidad de control a partir de la tabla anteriormente creada.

Unidad aritmética lógica

La ALU es la parte del procesador encargada de realizar todas las operaciones matemáticas y lógicas que estén implementadas. Nuestro set de instrucciones necesita tener tres operaciones lógico aritméticas:

- ANDS (operación lógica AND)
- ORS (operación lógica OR)
- LSL (desplazamiento a la izquierda)

Hay que recordar que la ALU realiza todas las operaciones al mismo tiempo pero, solo saca una como resultado. Esa operación la eligen los multiplexores los cuales son controlados mediante las señales de control de cada multiplexor.



Diseño de la unidad aritmética-lógica

Principios de la segmentación

Cuando hablamos de segmentación de un procesador nos referimos a dividir las operaciones que el procesador realiza en 'trozos' más pequeños. A esos 'trozos' se les denomina etapas y un procesador pequeño como este, normalmente consta de cinco etapas. Al ejecutar una instrucción, deberá pasar por todas las etapas que el procesador tenga. Esas etapas se conocen como:

1. Búsqueda de Instrucción.
2. Decodificación.
3. Búsqueda de Operandos.
4. Ejecución.
5. Almacenamiento de Resultados.

Operación de un procesador secuencial

1	2	3	4	5	6	7	...
Búsqueda1	Decodifica1	B.Operandos1	Ejecución1	Almacena1	Búsqueda2	Decodifica2	...

En un procesador NO segmentado las instrucciones se ejecutan secuencialmente tal como aparece en la imagen superior.

Segmentando un procesador no se disminuye el tiempo necesario que una instrucción necesita par completarse sino que se aumenta el número de instrucciones que se ejecutan por unidad de tiempo, disminuyendo así el tiempo medio de ejecución y aumentando el rendimiento efectivo del procesador. Una instrucción en un procesador segmentado se ejecuta de la siguiente forma:

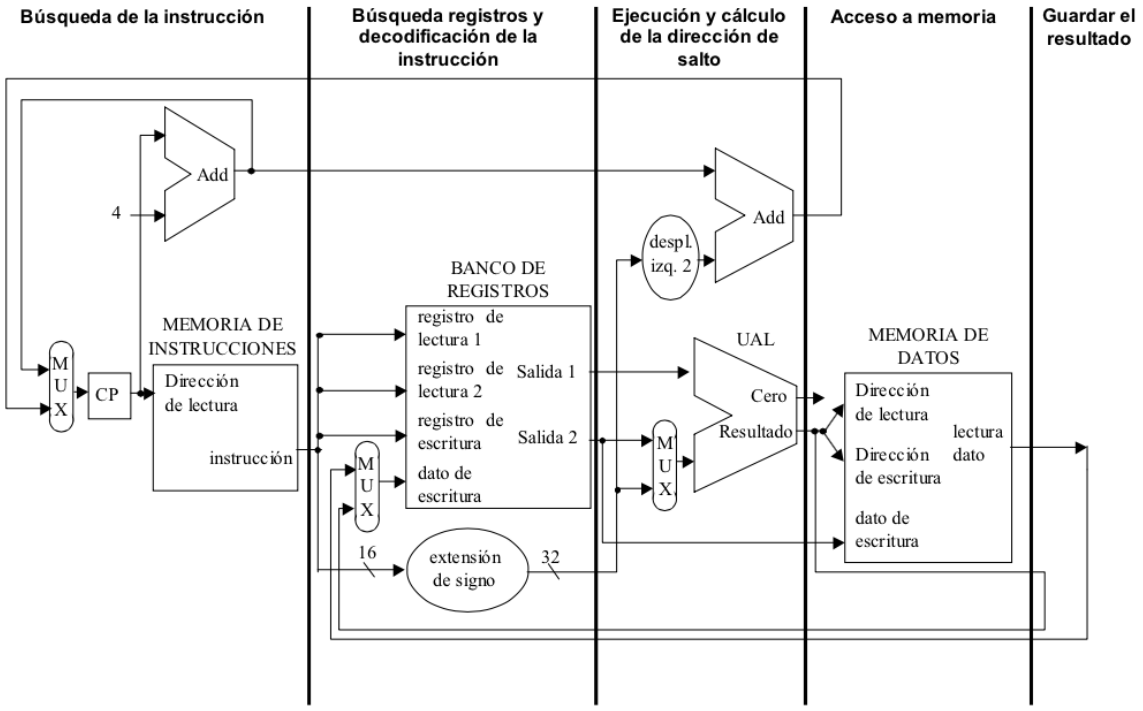
Operación de un procesador segmentado de 5 etapas

1	2	3	4	5	6	7
Búsqueda1	Búsqueda2	Búsqueda3	Búsqueda4	Búsqueda5	...	...
...	Decodifica1	Decodifica2	Decodifica3	Decodifica4	Decodifica5	...
...	...	B.Operandos1	B.Operandos2	B.Operandos3	B.Operandos4	B.Operandos5
...	...	...	Ejecución1	Ejecución2	Ejecución3	Ejecución4
...	...	...	...	Almacena1	Almacena2	Almacena3

Teóricamente, el tiempo que se reduce es proporcional al numero de etapas del procesador

$$T_{medio\_ejec.} = \frac{T_{ejec.\_todas\_las\_instruc.}}{N^{o}\_instruc.}$$

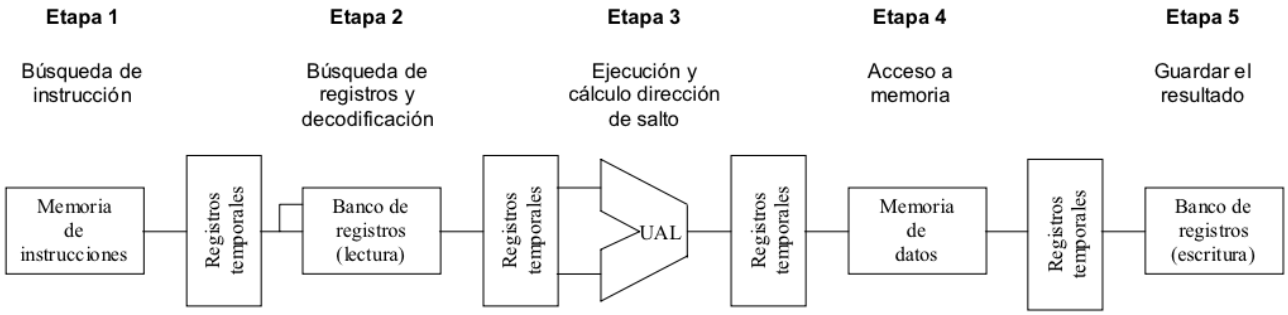
Etapas



Una vez segmentado el procesador en cinco etapas, tal y como se puede apreciar en la foto superior, se distingue claramente que operaciones se realizan en cada etapa.

- La primera etapa se dedica a buscar el código de la instrucción, es decir, a buscar la instrucción en la memoria de instrucciones.
- La segunda etapa, busca los registros especificados en los bancos de registros y genera las señales de control en base al código de operación de la instrucción.
- La tercera etapa, hace las operaciones matemáticas a través de la ALU.
- La cuarta etapa, accede a la memoria para la lectura de datos, siempre y cuando así lo haya especificado la instrucción.
- La quinta etapa, guarda todos los resultados obtenidos en los registros, en la memoria y ademas, es la etapa en la que se decide si ejecutar un salto o no.

Una vez diseñadas las etapas del procesador es importante tener en cuenta el nuevo camino de datos que van a seguir las instrucciones. Ahora, los datos pasarán de un registro interetapa a otro y de los registros a las partes del procesador correspondiente.



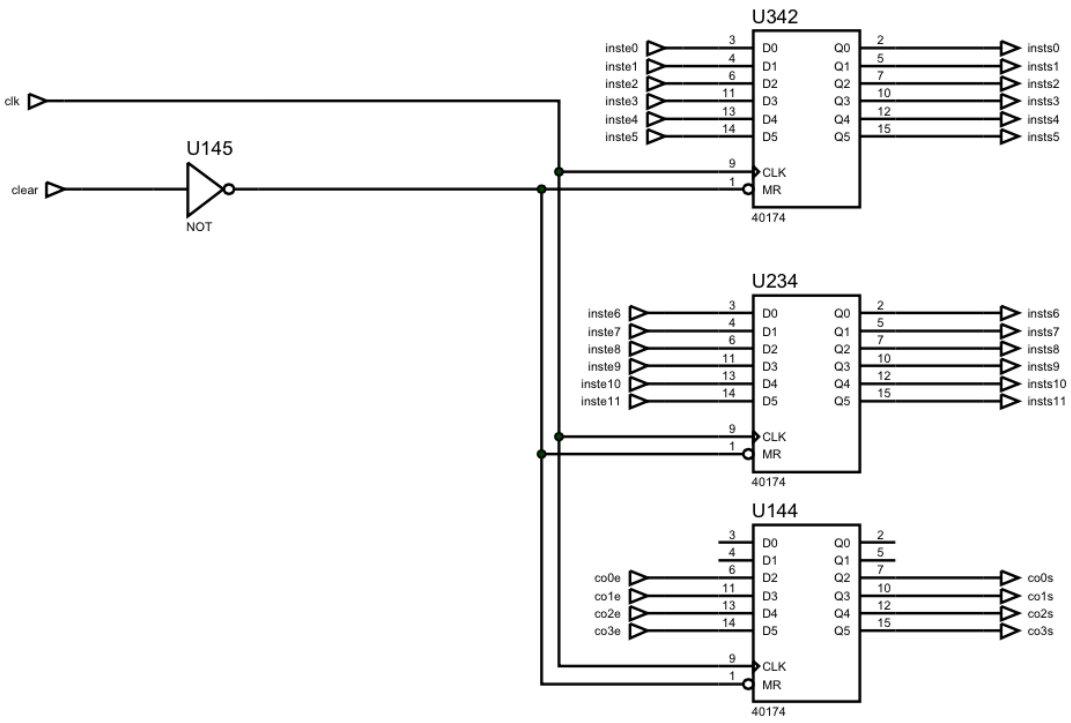
Los registros interetapa

El procesador tendrá cinco etapas dividas por cuatro segmentos interetapa.

1. Búsqueda de Instrucción.
2. Decodificación.
3. Búsqueda de Operandos.
4. Ejecución.
5. Almacenamiento de Resultados.

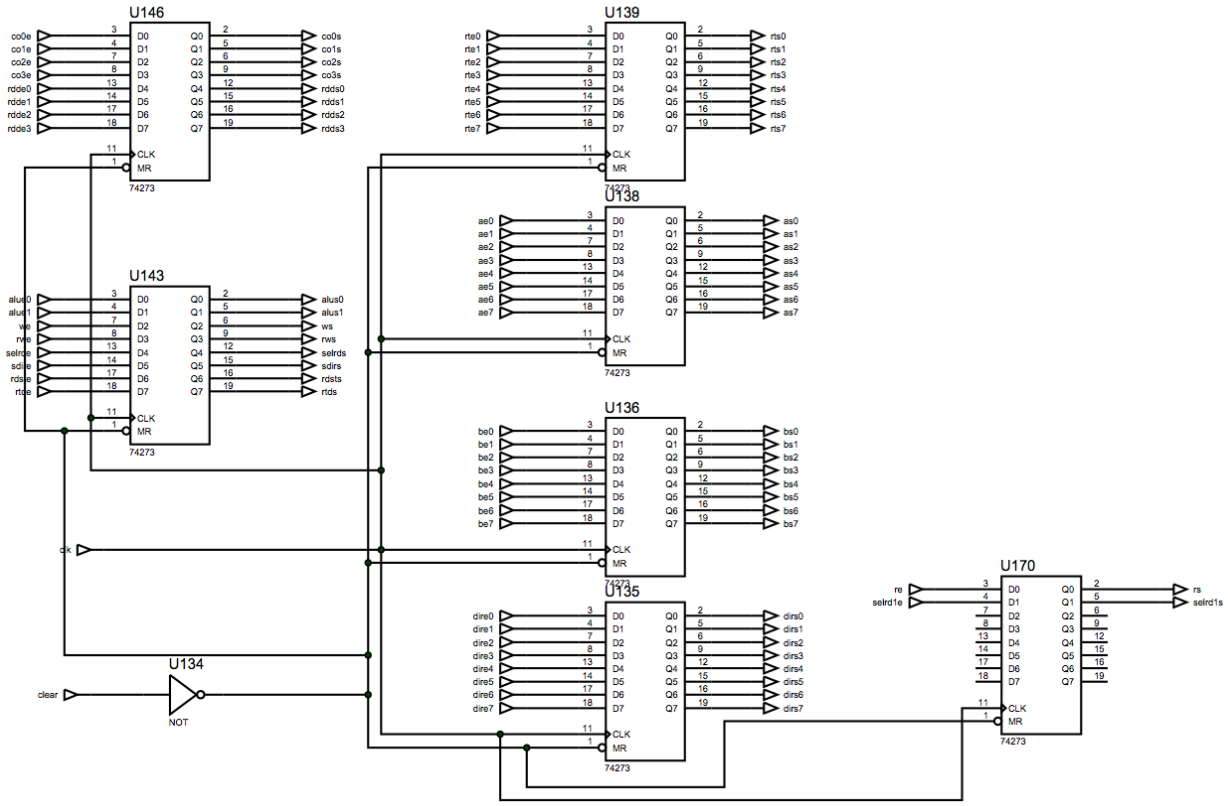
Se ha tomado la decision de enviar, ademas de los datos necesarios de cada registro, el código de operación de la instrucción en curso, para ayudar a depurar errores o explicar el funcionamiento a la hora de ejecutar la simulación. Los registros interetapa están formados por una serie de flip-flops que tienen en común la linea *CLK*, la cual ser usará para borrar su contenido e insertar, de esta forma, las famosas burbujas. A continuación se muestra la estructura de cada registro interetapa:

Primer registro



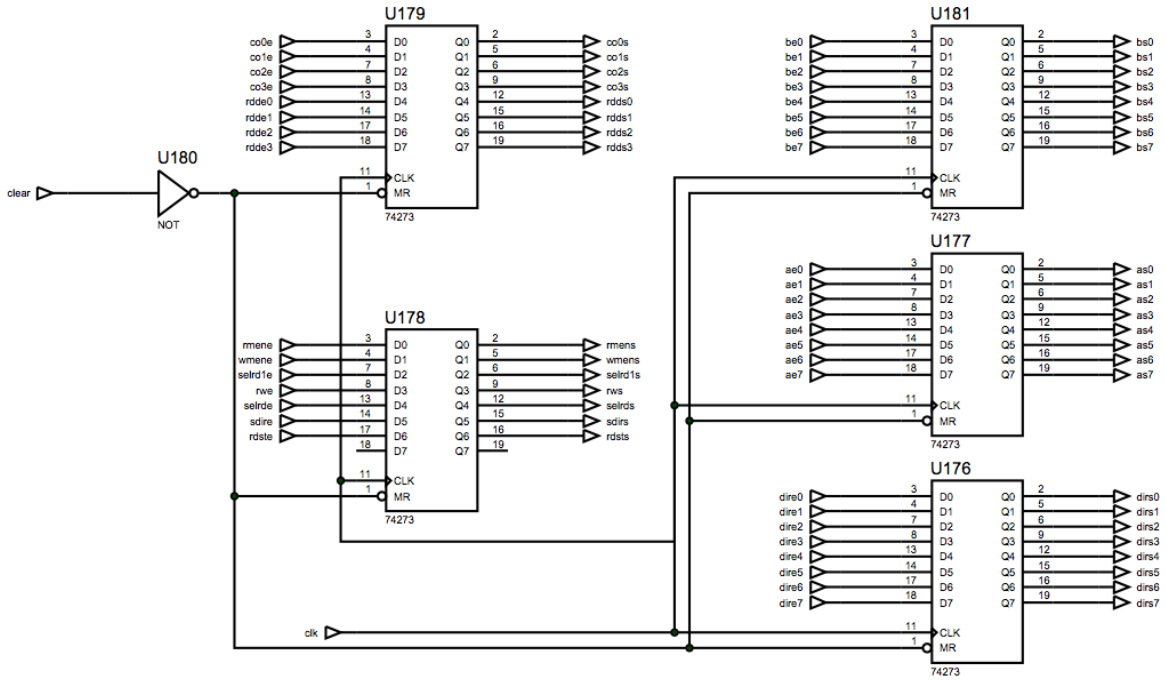
Representación interna del primer registro interetapa en Proteus (registro entre búsqueda de instrucción y decodificacion).

Segundo registro



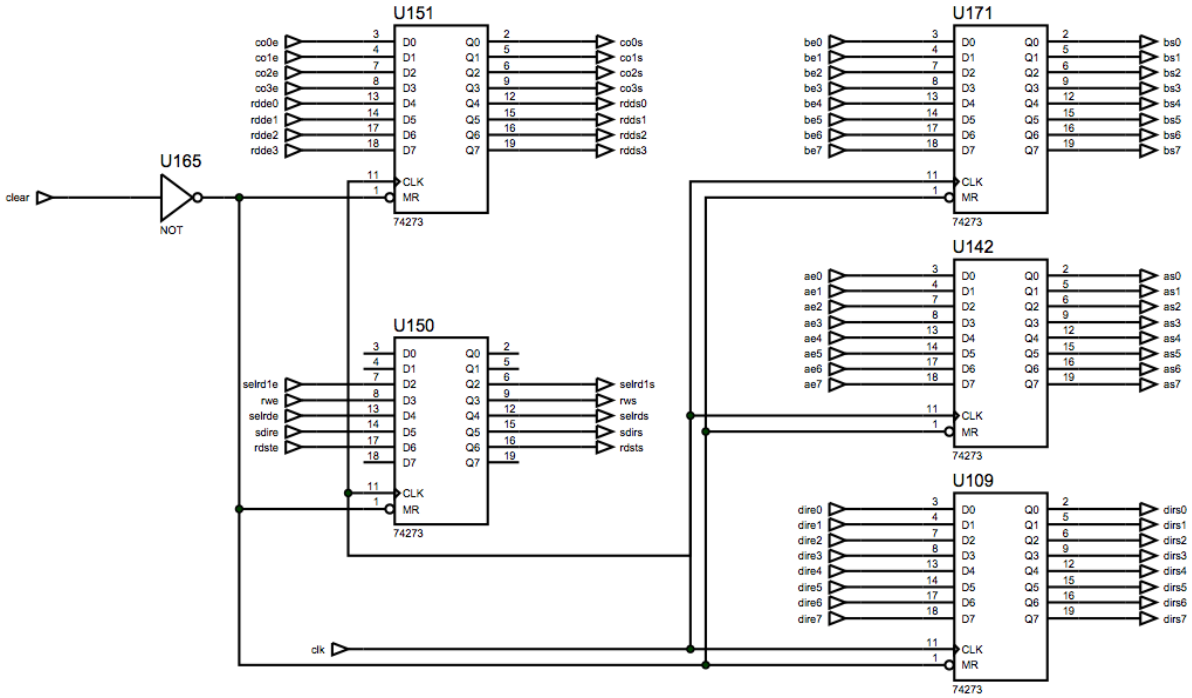
Representación interna del segundo registro interetapa en Proteus (registro entre decodificacion y búsqueda de operandos).

Tercer registro



Representación interna del tercer registro (registro entre búsqueda de operandos y ejecución) interetapa en Proteus

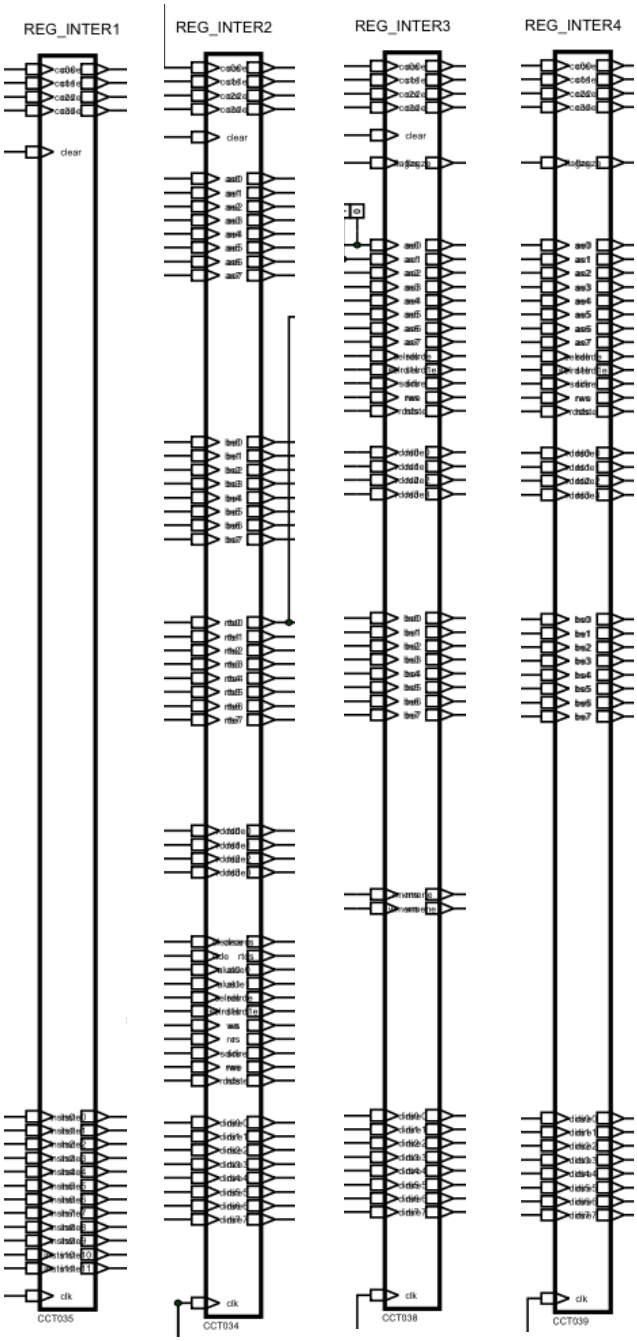
Cuarto registro



Representación interna del cuarto registro interetapa en Proteus (ejecución y almacenamiento).

El objetivo de un registro interetapa es almacenar el estado de las señales de control así como los datos necesarios para llevar a cabo la instrucción, en cada etapa evitando los diferentes conflictos que puedan surgir.

Todos los registros interetapa se han encapsulado en una macro con la siguiente estructura:



Burbujas

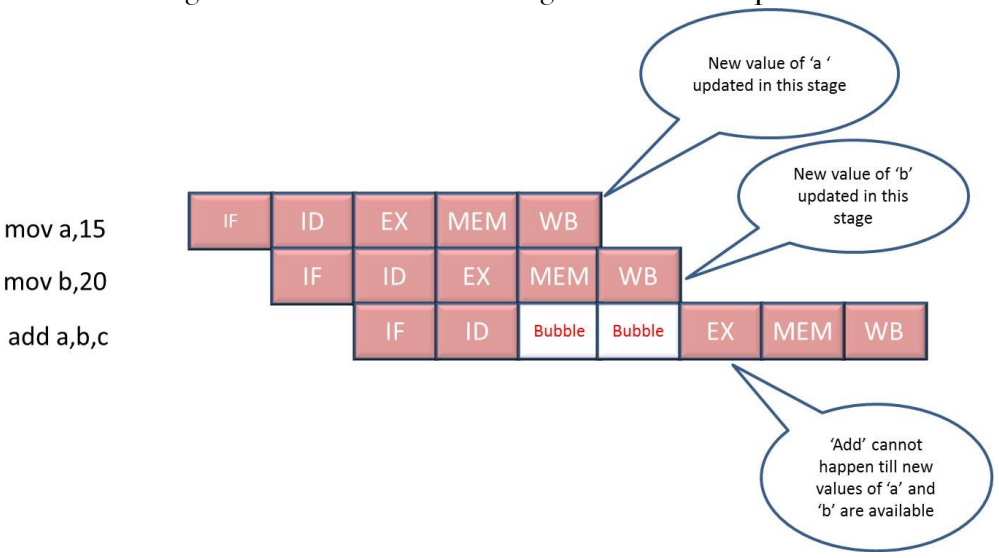
Una burbuja es la forma mas fácil de solucionar un conflicto, pero a su vez la menos eficiente. Simplemente se trata de meter instrucciones que no hacen nada en el cauce para que el procesador avance, ejecute esas instrucciones que no hacen nada como instrucciones normales y la instrucción que generaba el conflicto avance a la siguiente etapa. Las burbujas se deben meter hasta que la instrucción que generaba el conflicto acabe. En ese momento se dejan de meter burbujas, y se continua con la instrucción que se había quedado a la espera por causa del conflicto.

Pasos a la hora de insertar una burbuja:

1. Primero se detecta el riesgo.
2. Luego se detienen las instrucciones (se insertan burbujas) hasta que se resuelve el riesgo.

Como se mete una burbuja?

Una burbuja es un conjunto de señales inofensivas, es decir, que no afectan la ejecución del programa. En nuestro caso, una burbuja se indica mediante el bit *CLEARC* que pone a cero los registros internos de los registros interetapa.



Como se puede apreciar en la imagen, se están metiendo 2 burbujas para evitar que la operación *ADD* se ejecute cuando los datos *A* y *B* no han sido todavía cargados con sus correspondientes valores. La primera burbuja evita ejecutar *ADD* y al mismo tiempo se mueve *15* al registro *A*. La segunda burbuja evita ejecutar *ADD* y al mismo tiempo se mueve *20* al registro *B*. Una vez ya cargados, se ejecuta *ADD A, B, C*.

Conflictos

Hay circunstancias que pueden disminuir el rendimiento de un procesador segmentado debido a que provocan la detención del cauce, o ejecución de las instrucciones. Estas circunstancias se denominan riesgos o conflictos. Existen tres clases de conflictos:

- *Conflictos estructurales:* son detenciones producidas en el procesador por insuficiencia del hardware, debido a que una etapa no puede avanzar porque el hardware necesario está siendo utilizado por otra.
- *Conflictos por dependencia de datos:* surgen, principalmente, cuando una instrucción necesita los resultados de otra anterior, que todavía no los tiene disponibles, por no haberse terminado de ejecutar completamente.
- *Conflictos de control de flujo*: se deben a las instrucciones de control de flujo, en que no se puede leer la instrucción siguiente hasta que no se conozca su dirección.

Conflictos estructurales

Dado que es un procesador de arquitectura MIPS, el cual a su vez es RISC, el procesador dispone de dos memorias, una para datos y otra para instrucciones; así que no hay que tener en cuenta este tipo de conflicto.

Conflictos por dependencias de datos

Como ya se explicó anteriormente, este tipo de conflictos surgen cuando instrucciones tienen que acceder al mismo dato. Supongamos que tenemos dos instrucciones A y B que se ejecutan en ese mismo orden en un procesador segmentado. Las dependencias de datos que se puede diferenciar en tres clases:

RAW (read after write, es decir, lectura después de escritura): la instrucción B trata de leer un operando de un registro de datos antes de que la instrucción A lo haya escrito; de esta forma, B puede tomar un valor incorrecto de ese operando porque la instrucción A todavía no ha escrito su valor. Esta es el tipo de dependencia más común en los procesadores segmentados.

WAR (write after read, es decir, escritura después de lectura): la instrucción B trata de escribir en un registro antes de que sea leído por la instrucción A; por ello, A tomaría un valor incorrecto, ya que debería tomar el valor antes de modificarse. Este es un conflicto casi inexistente en este procesador ya que la escritura de los resultados en los registros de datos se realiza en los últimos segmentos y es difícil que esa escritura se adelante al segmento de lectura de una instrucción anterior.

WAW (write after write, es decir, escritura después de escritura): la instrucción B intenta escribir en un registro un operando antes de que sea escrito por la instrucción A. Dado que las escrituras se realizan en un orden incorrecto, el resultado final es el de la instrucción A cuando debería ser el de la instrucción B. Este conflicto sólo se produce en procesadores segmentados que escriben en más de una etapa, pero como este procesador realiza las escrituras en la ultima etapa, no tenemos que tener en cuenta este conflicto. De hecho, a la acción de escribir en la ultima etapa se la conoce como WB (Write-back).

Conflictos de control de flujo

Este tipo de conflicto se origina cuando el procesador detecta una instrucción de salto condicional y, dado que es condicional, no sabra si debe saltar o no hasta que la instrucción se ejecute. El problema reside en cuando saltar, y qué hacer en caso de que se deba saltar y romper el flujo normal de instrucciones. Si el procesador salta, debe conocer la dirección de salto de antemano y eliminar las instrucciones pendientes en las etapas anteriores mediante burbujas. Si no salta, no ocurre ninguna conflicto, y el contador de programa pasaría a la siguiente instrucción.

Para solucionar este problema, se ha tomado la decision de decidir si una instrucción debe ejecutar un salto o no, en la quinta etapa. De esta forma, si la instrucción tiene que saltar, se rellenan con burbujas las etapas anteriores y se evita, de esta forma, el conflicto por salto.

Conflictos entre instrucciones

Set de instrucciones

MOVIS	1	0	0	0	d	a	t	o					r	d	s	
MOVIT	0	0	0	1	d	a	t	o					r	d	t	
JRZ	0	0	1	0	d	i	r									
CLC	0	0	1	1												
LSL	0	1	0	0					r	s	d	s				
MOVMT	0	1	0	1	d	i	r						r	t		
ORI	0	1	1	0	d	a	t	o	r	s			r	d	s	
ANDS	0	1	1	1	r	t			r	s			r	d	s	
MOVMRT	1	0	0	1	d	i	r						r	d	t	

Set de instrucciones prueba

MOVIS	1	0	0	0	d	a	t	o					r	d	s	
MOVIT	0	0	0	1	d	a	t	o					r	d	t	
JRZ	0	0	1	0	d	i	r									
CLC	0	0	1	1												
LSL	0	1	0	0					r	s						
MOVMT	0	1	0	1	d	i	r						r	t		
ORI	0	1	1	0	d	a	t	o	r	s			r	d	s	
ANDS	0	1	1	1	r	t			r	s			r	d	s	
MOVMRT	1	0	0	1	d	i	r						r	d	t	

Set de instrucciones prueba

MOVIS	1	0	0	0	d	a	t	o					r	d	s	
MOVIT	0	0	0	1	d	a	t	o					r	d	t	
JRZ	0	0	1	0	d	i	r									
CLC	0	0	1	1												
LSL	0	1	0	0					r	s						
MOVMT	0	1	0	1	d	i	r						r	t		
ORI	0	1	1	0	d	a	t	o	r	s			r	d	s	
ANDS	0	1	1	1	r	t			r	s			r	d	s	
MOVMRT	1	0	0	1	d	i	r						r	d	t	

Set de instrucciones prueba

MOVIS	1	0	0	0	d	a	t	o					r	d	s	
MOVIT	0	0	0	1	d	a	t	o					r	d	t	
JRZ	0	0	1	0	d	i	r									
CLC	0	0	1	1												
LSL	0	1	0	0					r	s						
MOVMT	0	1	0	1	d	i	r						r	t		
ORI	0	1	1	0	d	a	t	o	r	s			r	d	s	
ANDS	0	1	1	1	r	t			r	s			r	d	s	
MOVMRT	1	0	0	1	d	i	r						r	d	t	

Conflictos por dependencia de datos en los registros

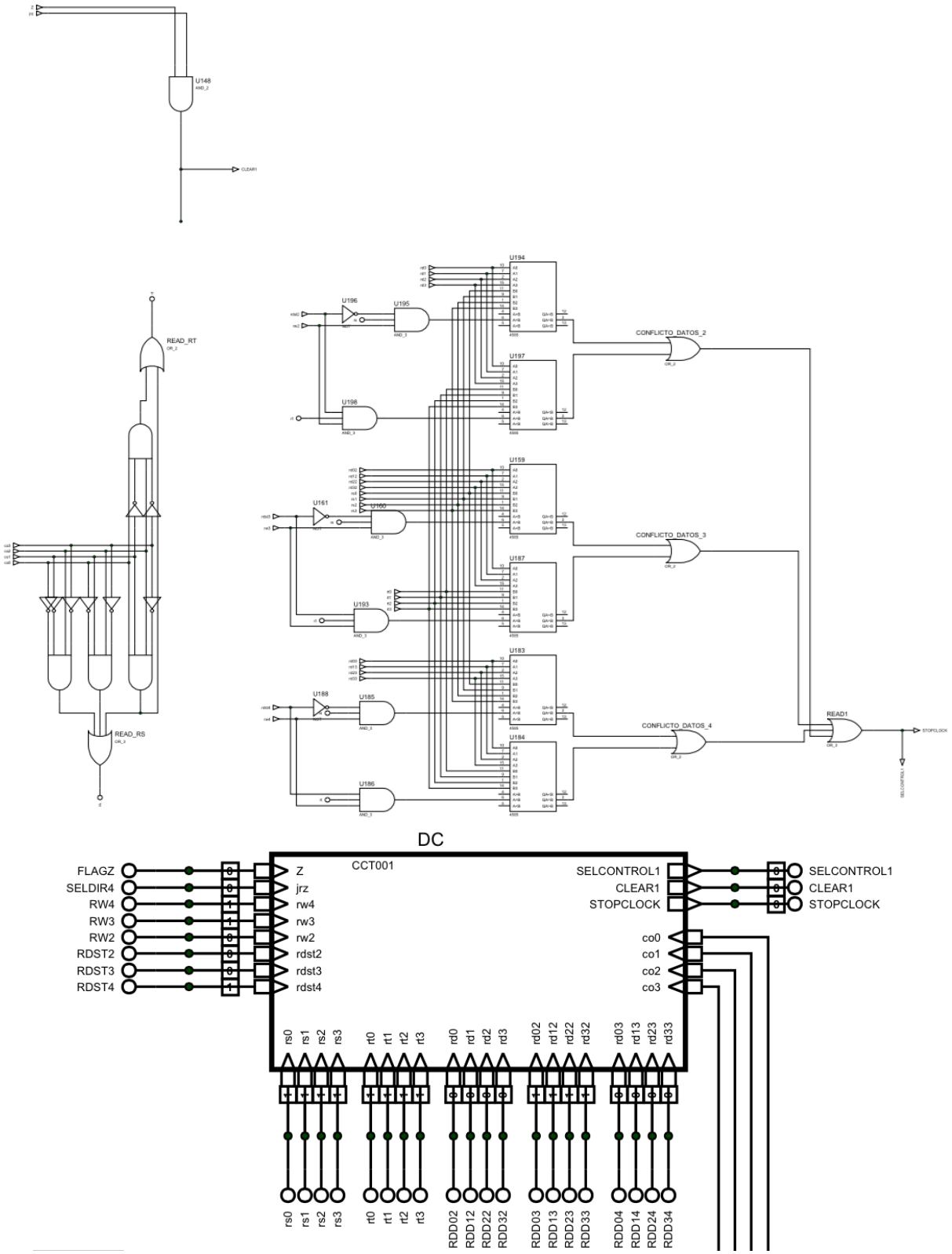
Como se puede apreciar en cada imagen, el set de instrucciones tiene multiple conflictos por dependencia de datos. Cada imagen representa un conflicto de datos en un registro o memoria determinada de tal forma que:

- Amarillo: representa las instrucciones que tienen conflicto de datos en el registro *RD* cuando *RDS* está seleccionado.
- Rojo: representa las instrucciones que tienen conflicto de datos en el registro *RD* cuando *RDT* está seleccionado.
- Verde: representa las instrucciones que tienen conflicto de datos en el registro *RS*.
- Verde: representa las instrucciones que tienen conflicto de datos en el registro *RT*.

Conflictos por dependencia de datos en memoria

También hay que destacar los conflictos que puede haber con las instrucciones que leen y escriben en memoria. Las instrucciones *MOVMT* y *MOVMRT* pueden generar este tipo de conflictos dependiendo el orden de ejecución de las mismas.

Unidad de detección de conflictos



Diseño de la unidad de detección de conflictos en Proteus

Instrucciones

Una instrucción es la unidad minima de acción que el procesador realiza y se constituye por una serie de campos

- Duración (en ciclos de reloj) (En el caso de los RISC, una instrucción un ciclo)
- Tamaño (RI).
- Numero de operandos.
- Tipo

En los procesadores monociclos cada instrucción se ejecuta en un ciclo de reloj. Esto genera una serie de consecuencias:

- Problema: El tiempo de ciclo de ejecución de instrucción se debe adaptar al de la instrucción más larga.
- Todas las instrucciones tardan lo mismo, aunque no lo necesiten, con lo cual se desaprovechan los recursos del ordenador en aquellas instrucciones que no necesiten tanto tiempo:

En los procesadores multiciclo, hay instrucciones que no utilizan todas las etapas o su fase de ejecución dura menos. A partir de este hecho se sacan varios puntos a tener en cuenta:

- Descomposición en etapas la ejecución de la instrucción. (Búsqueda instrucción, operandos, etc.)
  Cada etapa se ejecuta en un ciclo de reloj.
  Ciclo de reloj: Tan largo como la etapa de instrucción más larga.
  Varios tipos de instrucción en función de las etapas que utilizan.
- Cada instrucción se ejecutará en tantos ciclos como etapas tenga.Lo normal es que las unidades de control sean multiciclo.
  Permite mejorar las prestaciones del procesador usando técnicas como la SEGMENTACIÓN.

Set de prueba

Para probar el funcionamiento del procesador y verificar que funciona como debe se ha creado un set de prueba de instrucciones que engloba todas las instrucciones anteriores. El set de prueba tiene el siguiente orden:

1. MOVIS
2. MOVIT
3. LSL
4. MOVMT
5. ORI
6. ANDS
7. CLC
8. MOVMRT
9. JRZ

Setup0 y Setup1

La simulación de las instrucciones en Proteus requiere la utilización de dos ficheros auxiliares llamados Setup0.bin y Setup1.bin en los cuales están definidas las instrucciones que el procesador en Proteus simulara. Cada fichero contiene la mitad de la instrucción dividida en dos partes: la parte alta y la parte baja.

- Parte alta: contiene los ocho primeros bits de cada instrucción
- Parte baja: contiene los ocho bits restantes de cada instrucción

Contenido Setup0.bin

8212 4052 6d71 3092 2000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Contenido Setup1.bin

5081 0031 0444 0030 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Instrucciones: parte alta y parte baja

Del cada fichero setup0 y setup1 se obtienen los datos para obtener la instrucción, de tal manera que se junta la parte al ta y la parte baja, se obtiene la instrucción a ejecutar en formato hexadecimal. A continuación, se listan las instrucciones que el procesador ejecutara en el test de prueba en el mismo orden de ejecución.

8250

1281

4000

5231

6D04

7144

3000

9230

2000

Instrucciones traducidas a ASM [base16]

Una vez obtenidas las instrucciones en formato hexadecimal, se convierten las instrucciones a ensamblador o código maquina, para que se entiendan mejor.

Código OP	8bits	4bits	4bits
MOVIS	25	0	
MOVIT	28	1	
LSL	0		
MOVMT	23	1	
ORI	D	0	4
ANDS	1	4	4
CLC			
MOVMRT	23	0	
JRZ	00		

Instrucciones traducidas a ASM [base2]

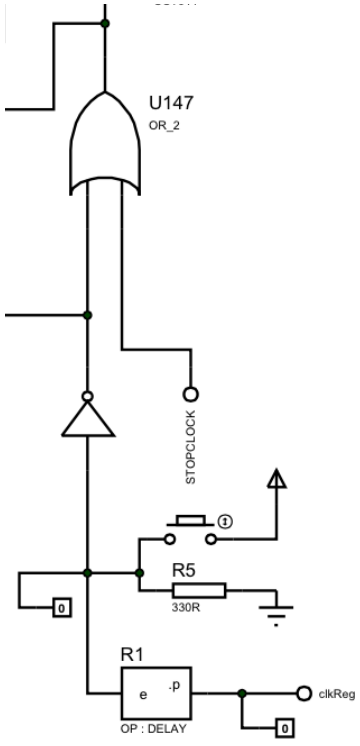
Para comprobar que el camino de datos y los propios datos de la instrucción son correctos es recomendable traducir la instrucción a binario y comprobar en Proteus que se está ejecutando correctamente.

Código OP	8bits	4bits	4bits
1000	0010	0101	0000
0001	0010	1000	0001
0100		0000	
0101	0010	0011	0001
0110	1101	0000	0100
0111	0001	0100	0100
0011			
1001	0010	0011	0000
0010	0000	0000	

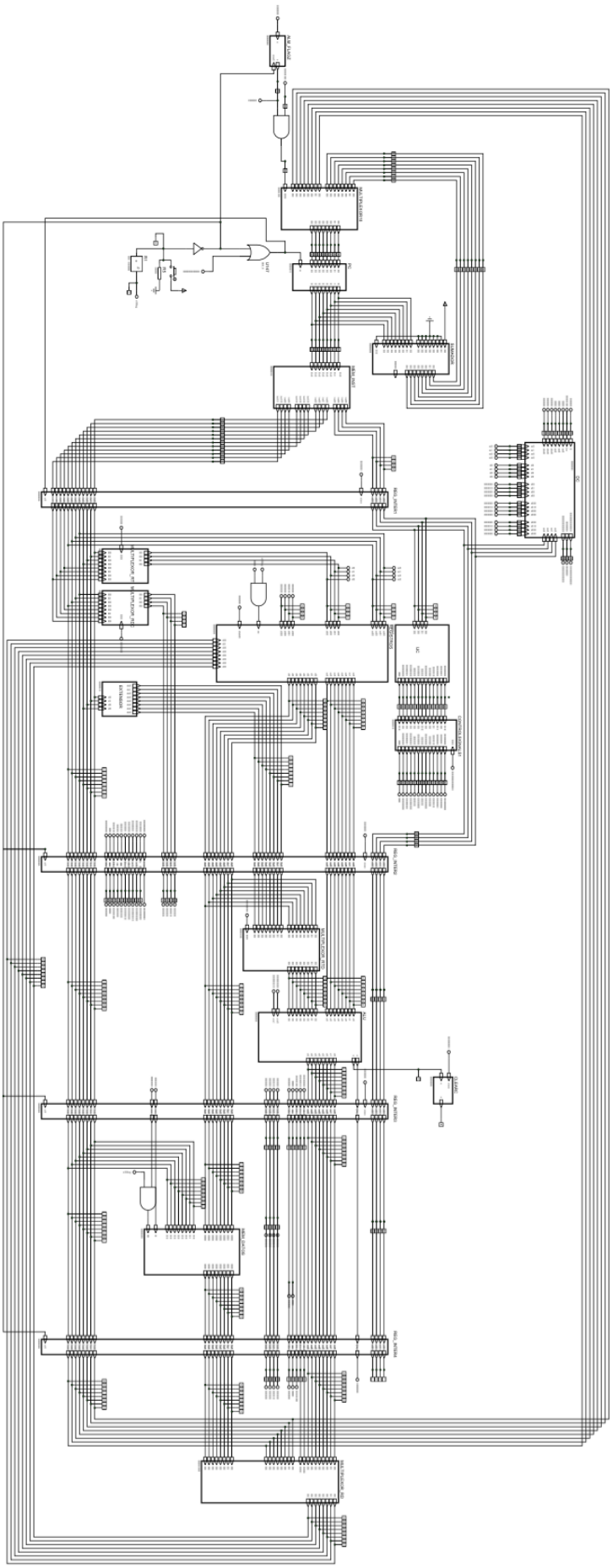
Problemas surgidos

Uno de los mayores problemas que hemos tenido a la hora de diseñar el procesador segmentado ha sido desafiar la señal de reloj que usaban los registros para evitar que las operaciones escribieran y leyeran al mismo tiempo. La primera solución que intentamos, fue añadir una numero par de inversores a la señal de reloj que llegaba a los registros. Pero no funcionó. Así que probamos la solución fácil: añadir un segundo reloj, pero este con un cierto desfase. Al parecer, esta solución no cumplía con los estándares de un procesador así que tuvimos que buscar en Proteus un delayer para desafiar la señal de reloj de la forma correcta. Al final, tuvimos dos señales de reloj que funcionaban a la misma frecuencia pero una de ellas iba desfasada.

Solución



También es cierto que hemos tenido mas problemas como, la representación de los valores en Proteus, algunas señales de control no se actualizaban correctamente, … Y otros no tan importantes como que la señal clear de los flip-flop se activaba por nivel bajo.



Bibliografía

- www.infor.uva.es/~bastida/Arquitecturas%20Avanzadas/Segment.pdf
- profesores.elo.utfsm.cl/~tarredondo/info/comp-architecture/paralelo2/ProcesadorSegmentado.pdf
- personales.upv.es/pabmitor/acso/FILES/ArqComp/CST/ArqComp%20t4.pdf
- studies.ac.upc.edu/EUPVG/ARCO_I/tema2.pdf