Instituto Tecnológico de Buenos Aires

22.15 Electrónica V

Trabajo práctico 2: Procesador EV20

Grupo 3

Chiocci, Ramiro Antonio	45132
Goytía, Agustín	56023
Kipen, Javier	57345
Parra, Rocío	57669
Reina Kiperman, Gonzalo	56102

Profesores

Rodríguez, Andrés Carlos
Wundes, Pablo Enrique

Presentado: 11/06/2020

Índice

1. Introducción

En el presente informe se detallará el diseño de la arquitectura de un procesador programable RISC llamado EV20. En primer lugar, se explicará como se implementó la estructura genérica del procesador. Luego se discutirán los cambios propuestos con el objetivo de agregar modo de direccionamiento indexado, y por último se hablará de los programas utilizados para probar el hardware, y de las conclusiones del trabajo.

1.1. El procesador EV20

El procesador EV20 tiene una microarquitectura basada en microcódigo, donde cada instrucción corresponde a una única microinstrucción. Cuenta con una pipeline de 5 etapas (*fetch*, *decode*, *operand*, *execute*, *retire*), y presenta una arquitectura Harvard (es decir, tiene memorias y buses separados para datos e instrucciones).

El procesador EV20 opera con 34 registros de 16 bits, mapeados en un *register bank* como $R_0 - R_{34}$, a saber:

$R_0 - R_{27}$: registros de propósito general

$R_{28} - R_{29}$: puertos de entrada PI_0 y PI_1

$R_{30} - R_{31}$: puertos de salida PO_0 y PO_1

R_{34}**:** registro W (*working register*)

Se considera que si el programa intenta escribir el registro R_{35} (que no existe), se está queriendo indicar que no se debe guardar el resultado de la operación. Esto es una situación similar a un *nop* (*no operation*), con la salvedad de que para algunas operaciones podría verse modificado el carry. No es válido referenciar a registros R_{32}, R_{33}, o R_{36} o mayor, así como intentar leer del registro R_{35}, o intentar escribir un puerto de entrada.

En cuanto al registro W, si bien es un registro de 16 bits como cualquiera de los genéricos, tiene la particularidad de ser el único que puede utilizarse para escribir una palabra a la memoria, o para leer desde ella. Además, se utiliza para la mayoría de las operaciones aritméticas-lógicas, tanto como operando como registro destino del resultado.

Estos registros interactúan con la ALU a través de tres buses: A y B, que funcionan como operandos, y C, donde se obtiene el resultado de la operación. Cada microinstrucción (o instrucción, para algunos casos específicos) indica qué valores deben tomar los buses A y B, y dónde debe guardarse el resultado obtenido en C, así como la operación que los relaciona. No en todas las instrucciones el valor de todos los buses será relevante (por ejemplo, para *set carry* no se utiliza ninguno de los tres).

Este procesador es una evolución de versiones anteriores EVxx, donde "xx" son los dos últimos dígitos del año en que se desarrolló. En particular, hereda el set de instrucciones y el esquema de funcionamiento general del EV16, con dos diferencias fundamentales:

- utiliza registros de 16 bits, en lugar de los de 8 bits de su predecesor, con lo cual la palabra de instrucción se hace 8 bits más grande
- posee un stack de direcciones de retorno de cuatro niveles, que le permite anidar hasta cuatro subrutinas

2. Set de instrucciones

2.1. ISA original (EV16)

Las instrucciones del EV20 pueden separarse entre los siguientes tipos:

- De salto absoluto: incondicional o condicional (puede ser según si el carry es 1, si el registro *W* es 0 o si *W* es negativo)
- De manejo de subrutinas: salto relativo incondicional a subrutina, o retorno desde subrutina
- Lectura o escritura de memoria
- Operaciones lógicas: and, or y complemento a uno. Todas utilizan el registro *W* como uno de los operandos, y el otro puede ser un registro de propósito general o una constante. El resultado se guarda en uno de los operandos.
- Operaciones aritméticas: suma con carry. Puede utilizar los mismos operandos que las operaciones lógicas. El resultado se guarda en uno de los operandos.
- Operaciones de *move*: copiar una constante de un lugar a otro. Los lugares de fuente y destino pueden ser la memoria de datos, la instrucción (una constante), el registro *W*, los puertos de i/o o los registros de propósito general. La memoria sólo puede interactuar con *W*, así como las constantes provenientes del programa, pero los registros de propósito general pueden copiarse entre sí o a los puertos de i/o.
- Set carry / clear carry

2.2. Extensión para constantes de 16 bits

Las únicas instrucciones que necesitan modificarse para utilizar registros de 16 bits son las que operan con constantes, dado que su valor está incluido en la instrucción.

Existen cuatro instrucciones que cumplen con este criterio: "*MOK W, #K*", "*ANK W, #K*", "*ORK W, #K*" y "*ADK W, #K*" (cargar *W* con el valor de la constante, o realizar un *and* bit a bit, un or bit a bit o una suma con carry entre *W* y la constante, en ese orden). Para todas ellas, los primeros 6 bits indican la operación a realizar, y los restantes bits, el valor de *K*. Por lo tanto, para poder operar con registros de 16 bits, estas instrucciones deben tener 22 bits (6 indicando la operación y 16 el operando).

Esto implica que todas las instrucciones para el EV20 deben ser de 22 bits. Para las demás instrucciones, que no requieren estos 8 bits extra, el valor que tomen es irrelevante, y se decidió fijarlo en 0.

Sin embargo, como estos bits que se agregan no afectan a las señales de control que debe proveer la *micro-instruction ROM*, no es necesario modificar el tamaño de sus palabras para implementar esta modificación.

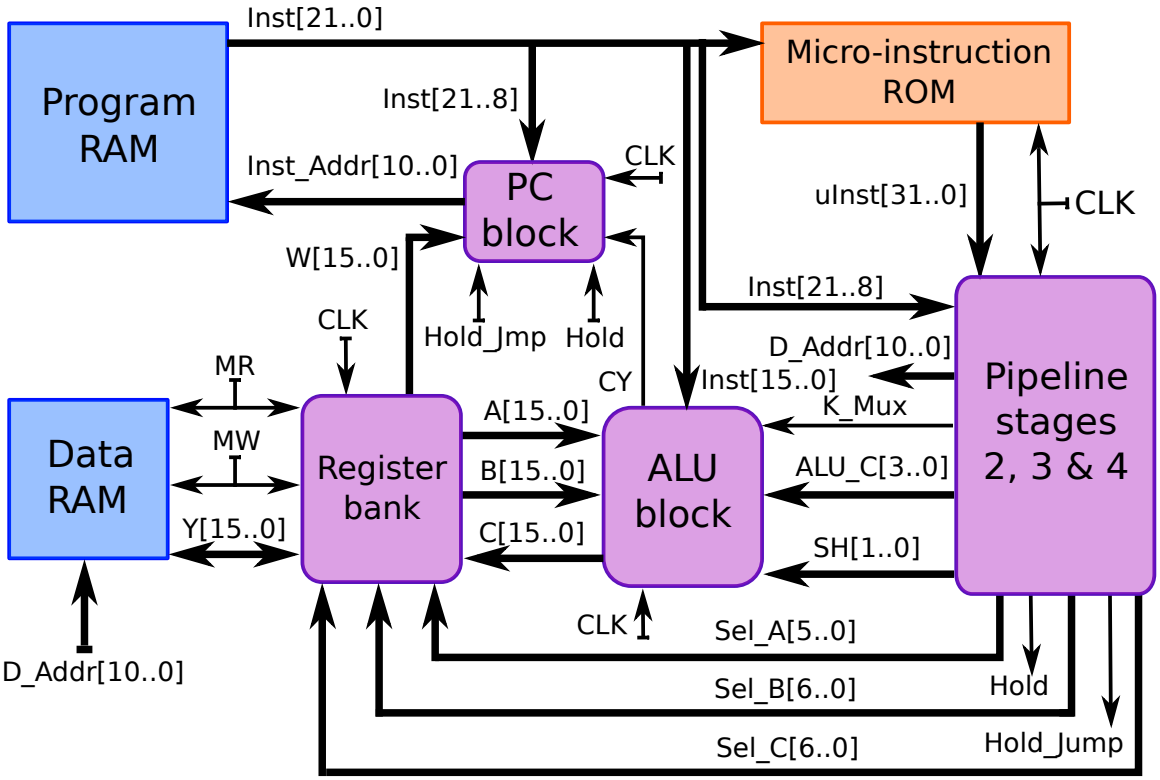

Figura 1: Diagrama de bloques del EV20

3. Control Store y Microinstrucción

3.1. Estructura de las microinstrucciones

Para decodificar las microinstrucciones se utilizó una memoria a modo de LUT. Dado que con los primeros 7 bits se puede determinar que instrucción es, se diseñó una memoria con 7 bits de Address.

Cada palabra de microinstrucción consiste en los siguientes campos (Tabla 1)

Campo	Cantidad de bits
I	2
ALUC	4
SH	2
KMx	1
M	2
B	6
C	6
T	7
Total	30

Tabla 1: Campos de la microinstrucción

Los campos son los propuestos por el enunciado, exceptuando el campo I. Éste se explica en la sección de la ampliación del ISA.

Por último, se completó la memoria manualmente con los valores descritos en la tabla A del enunciado.

3.2. Modificaciones al esquema general

Dado que las memorias en quartus son obligatoriamente sincrónicas, se tomó que la entrada al address sea la instrucción antes del latch. De esta manera se puede mantener el pipeline con una memoria sincrónica.

Además, ciertas instrucciones tienen campos de la microinstrucción dentro de la misma. Por ejemplo, la dirección de lectura de memoria está incluida en la instrucción. Para algunos casos, se tiene que decidir si se toma de la memoria o de la instrucción, como el campo C. Para eso se explicitaba una señal en el diagrama del enunciado, pero la instrucción ADW no estaba contemplada. Esto fue agregado con lógica simple.

4. Banco de Registros y ALU

4.1. Banco de Registros

El banco de registros no ha sido modificado con respecto al sugerido en el EV16, salvo aquellas necesarias para implementar funcionalidad extra, que será desarrollado más adelante.

4.2. Control de la ALU

La ALU es la parte que realiza las operaciones del microcontrolador. La implementación del bloque de control de la ALU se muestra en la Figura 2.

La ALU propiamente dicha es un circuito combinacional que recibe 3 valores: Un valor A proveniente de un latch (bus), un valor B proveniente de un latch (bus), y un carry, que proviene también de un latch. La salida de la ALU puede ser modificada por un circuito SHIFT, y finalmente llega a un latch de salida (bus). Es decir, las entradas de la ALU se encuentran latcheadas (en latch A, latch B y latch C) y la salida (modificada por el circuito SHIFT) se almacena en el Latch C. Ante un flanco positivo del clock, se actualizan los valores de los Latches. El valor de entrada del Latch B está dado por un bus externo al bloque de control de la ALU (que estará conectado al bus de salida B del banco de registros). El valor de entrada del Latch A depende de K_{Mux} (entrada del bloque). Dependiendo del valor de dicha señal, la señal de entrada del Latch A estará determinado por uno de dos buses de entrada del bloque de control. Si $K_{Mux} = 0$, entonces el latch A está conectado (de manera externa al bloque) al bus de salida A del banco de registros. Caso contrario, se determinará por una entrada auxiliar del bloque que permite cargar una constante proveniente de una instrucción. Con respecto al carry, el circuito asociado al latch de carry se encarga de preservar el valor del carry entre operación y operación, a menos que se ejecuten las instrucciones que explícitamente modifican el carry a 1 o 0. Las salidas de la ALU son la salida del bloque Latch C, y la salida del bloque Carry, que representan el resultado de la última operación con el carry asociado.

Las operaciones que realiza la ALU y el SHIFT se muestran en las Tablas 2 y 3 respectivamente.

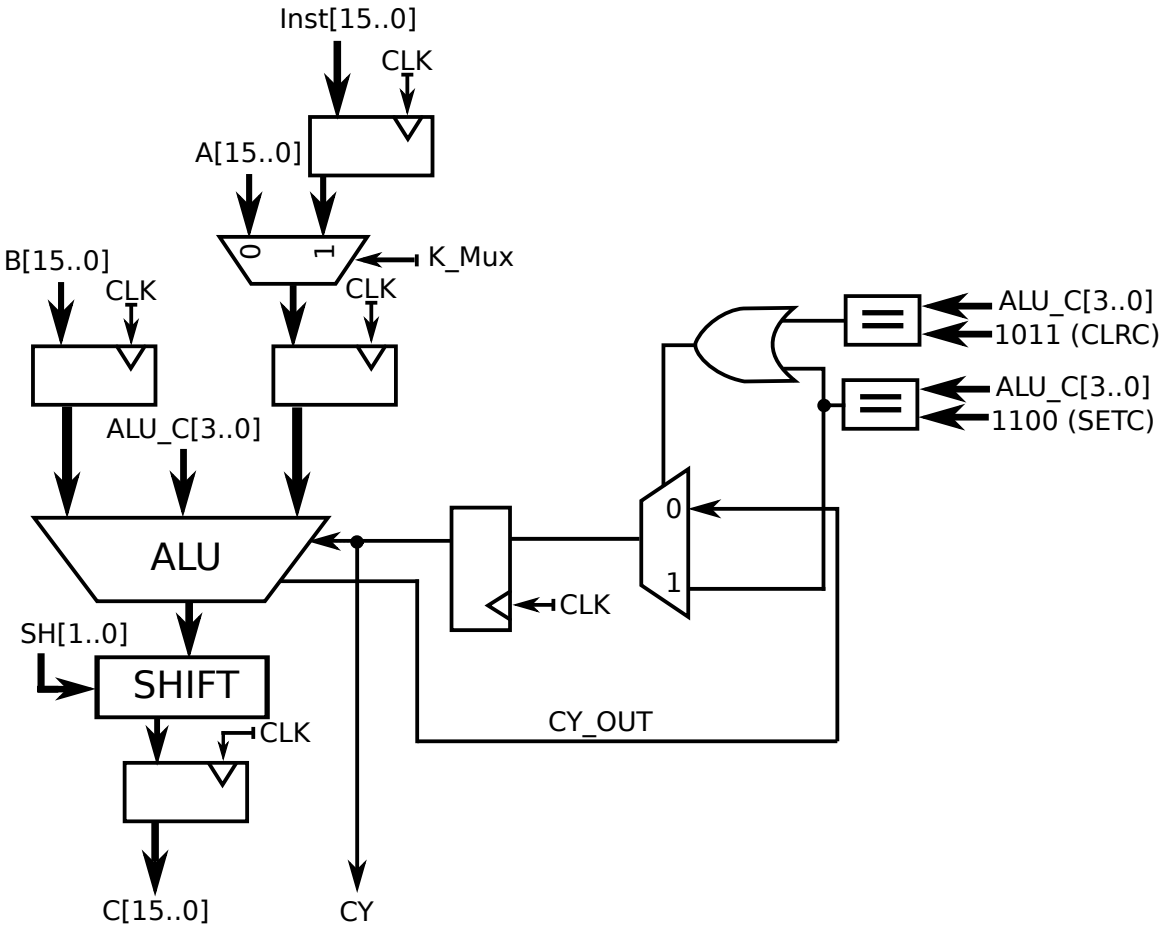

Figura 2: Bloque de control de la ALU

Entrada ALU	Salida	Carry Out
0000	A	C_{in}
0001	B	C_{in}
0010	/A	C_{in}
0011	/B	C_{in}
0100	A+B	Carry de suma
0101	A+B+Cy	Carry de suma
0110	A OR B	C_{in}
0111	A AND B	C_{in}
1000	0	C_{in}
1001	1	C_{in}
1010	0xFFFF	C_{in}
1011	0	0
1100	0	1
Otro	0	C_{in}

Tabla 2: Operaciones de la ALU. Se encuentran también reflejados los cambios producidos por el circuito de carry

Entrada SHIFT	Salida
00	No Shift
01	Shift right 1 bit
10	Shift left 1 bit

Tabla 3: Operaciones del circuito SHIFT

5. Pipeline

5.1. Etapas de pipeline

Se siguió el esquema especificado en el enunciado de 5 etapas:

- Fetch: En esta etapa se traen de la memoria el programa y se decodifica la microinstrucción asociada. Además, en casos de saltos con dependencia mantiene la instrucción hasta que se resuelvan los conflictos.
- Decode: La instrucción y la microinstrucción asociada pasan por un latch y se escogen con ellas el contenido de los buses A y B. Además, cuando hay problemas de dependencias, esta etapa se frena e inserta operaciones nulas en las siguientes etapas hasta que no haya dependencias.
- Operand: Llegan las entradas y la configuración a los bloques de ALU y shift
- Execute: El resultado del bloque anterior pasa por un latch al bus C
- Retire: Se guardan los valores del bus C en el correspondiente registro

5.2. Adaptaciones del esquema en las etapas de fetch y decode

Al basarse en el esquema del enunciado, se contempla que todas las memorias utilizadas son asincrónicas. Esta asunción no es correcta a la hora de utilizar los bloques de quartus, ya que los mismos cuentan obligatoriamente con una señal de clock. Las modificaciones que se hicieron para adaptar el esquema fueron las siguentes:

- Para la memoria de uIROM, se utiliza de dirección de entrada la instrucción antes de pasar por el latch. De esta manera, es sincrónica y cumple su función con el flanco ascendente.
- Las memorias de Data y Program se las actualiza sincrónicamente con la señal del clock negada, asegurando que entre flancos positivos se actualice.

5.3. Lógica del bloque UC-1

El funcionamiento del bloque UC-1 es muy sencillo: Cuando la entrada hold se pone en 1, evita lecturas y escrituras de memoria solicitadas por la etapa 2 del pipeline, e inyecta "NOPs" en la etapa 3. Para inyectar "NOPs", copia el valor de la etapa 2 a la etapa 3, pero pone el valor de control de la ALU a 0 para evitar que se modifique el carry, pone el valor del campo C a 35, indicando que el resultado no debe ser guardado, y no copia el campo T, dejando T vació en la etapa 3, para evitar que se generen dependencias inexistentes. El valor del resto de los campos no resulta importante, y son simplemente copiados.

6. Resolución de conflictos de dependencias

Los conflictos y dependencias son detectados por el bloque UC-2 (Figura 3) 3. Este bloque se encargará de generar dos señales:

- hold, que detendrá el pipeline y el avance del PC ante una dependencia.
- hold_jump, que detendrá el avance del PC sin detener el avance del pipeline (permitiendo que se resuelvan los conflictos de salto). Esto se debe a que los saltos se resuelven en la etapa 1, y si hay dependencia con la etapa 2, la misma debe poder avanzar por el pipeline.

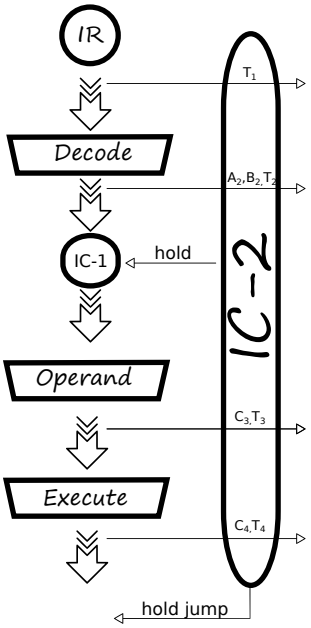

Figura 3: Control de dependencias

Para poder resolver dependencias, el bloque UC-2 recibe T_1, T_2, T_3 y T_4, asi como también recibe el valor de los buses A_2, B_2, C_3 y C_4.

6.1. Análisis del campo de microinstrucción Type

El registro Type de las microinstrucciones brinda información acerca de los registros que lee o modifica cada instrucción. A partir de él se puede expresar la lógica para el frenado del pipeline. Se utiliza la siguiente notación

$$T_i = \left[b^i_J, b^i_{CW}, b^i_{CR}, b^i_{RW}, b^i_{RR}, b^i_{WW}, b^i_{WR}\right]$$

Donde T_i es el campo Type de la microinstrucción en la etapa iésima del pipeline. b representa en orden a cada bit de T_i y los subíndices representan que registro y como interacciona, en orden. Los registros son el PC (J), carry, registro del banco y el registro de trabajo. Los modos de interacción son escritura o lectura (W o R).

6.2. Dependencia RAW en el registro W

$$hold = b^2_{WR} \cdot (b^3_{WW} + b^4_{WW})$$

El registro especial W ("Working Register") posee una estructura de 16 bits en donde se realizan la mayoría de las operaciones lógicas y aritméticas del procesador, las cuales complementa con acceso de lectura y escritura a la memoria de datos que utiliza para tales fines. Se definen instrucciones que involucran dependencias de escritura y lectura posterior (RAW) como puede ser una operación inconclusa de escritura del registro mediante datos de la memoria (MOM W,Y), seguida de una operación lógica que necesita del resultado anterior (ANK W,#K), lo que genera un conflicto que desencadena resultados inválidos. Los pasos para detectar y corregir este comportamiento se explican a continuación.

- El bloque UC-1, es el único que puede mediante una señal de entrada (hold), detener las operaciones entrantes en el módulo de decodificación del "pipeline", lo que sugiere que el código de la nueva instrucción se evalúe en este momento.
- El pipeline solo posee dos módulos que pueden presentar un tiempo de ejecución variable según la complejidad de las tareas realizadas. El módulo de ejecución depende directamente de la complejidad de las tareas a realizar por la ALU y, el módulo de finalización que depende del tiempo de llegada del resultado de la operación al destino especificado (memoria de datos, registro o puerto de salida).
- El bloque UC-2 se encarga de evaluar las operaciones activas en los módulos especificados con anterioridad para determinar si existe la posibilidad de solapamiento de tareas en el pipeline. En caso afirmativo, activa la señal de parada (hold) evitando el avance de las instrucciones. Este proceso se lleva a cabo mediante una evaluación lógica del código de las instrucciones activas.

6.3. Dependencia RAW en el banco de registros

$$hold = b^2_{RR} \cdot \left[b^3_{RW} \cdot (A_2 == C_3 + B_2 == C_3) + b^4_{RW} \cdot (A_2 == C_4 + B_2 == C_4)\right]$$

Además del registro W, existen otros 34 registros de 16 bits que están vinculados según su funcionalidad tanto a los buses de salida (A y B) como al de entrada (C). Como se explicó en la sección anterior, en este contexto una lectura de cualquiera de ellos cuando se encuentran en proceso de escritura puede generar conflictos de dependencia (RAW) como consecuencia de la interconexión mediante los buses. Podemos describir la situación de la siguiente manera.

- La escritura de los registros se realiza en el último ciclo del pipeline mientras que la lectura la hacemos en el tercero ("operand") lo que genera una dificultad si los tiempos de ejecución de cada módulo no se mantienen constantes. Al igual que el caso anterior, las instrucciones críticas son las ubicadas en el tercer y cuarto módulo del pipeline.
- IC-2 vuelve a ser el módulo encargada de detectar y corregir la situación pero, a diferencia del caso anterior, una imagen de los buses de entrada y salida es requerida para la evaluación lógica a realizar sobre los códigos de las instrucciones. Esto se da para evita que la lectura contenga los datos del registro previos a la escritura del mismo.

6.4. Dependencias del carry

En cuanto al carry, las dependencias de lectura luego de escritura no son relevantes, ya que son de etapas contiguas del pipeline. Esto lleva a que no haya problemas de dependencia.

Sin embargo, para el caso del salto condicional con el valor del carry, se puede presentar un caso en el cuál se tenga que frenar por una operación el pipeline para obtener el resultado del carry. Esto es cuando se tiene una instrucción que modifica el carry seguido por JCY.

6.5. Análisis de caso especial con escritura desde memoria

La instrucción de lectura de memoria tiene la particularidad de tener solo 3 etapas, modificando el valor del registro de trabajo.

Existen instrucciones de 5 etapas en las cuales escriben en su última etapa el registro de trabajo. Si en el código se escribe alguna de ellas seguida de una lectura de memoria, la lectura de memoria se hace antes que la instrucción en cuestión si no se frena el pipeline.

La lógica para contemplar esta dependencia es simple, si alguna de las microinstrucciones en las últimas 2 etapas modifican el registro de trabajo y en la etapa 2 se encuentra una lectura a memoria, se activa también la señal de hold.

6.6. Dependencias en los saltos

En los casos de los saltos condicionales, puede suceder que la instrucción que varíe el campo que produce el salto condicional no haya terminado de ejecutarse. Además, a diferencia de las demás instrucciones, el pipeline debe ser frenado cuando hay una dependencia de este tipo en la segunda etapa, ya que sino trae una instrucción inválida de memoria.

Para estos tipos de dependencia se utilizó otra señal denominada hold_jump. Esta es activada cuando hay un salto en la segunda etapa, que tiene lectura de carry o del registro de trabajo, y alguna de las operaciones en el pipeline modifica el mismo registro.

7. PC, implementación de saltos, manejo de subrutinas y condiciones de salto

Estos bloques, si bien son sencillos desde el punto de vista de los componentes que los forman, tienen cierta complejidad debido a su funcionamiento y las restricciones presentes. Por este motivo, se incluye un análisis detallado.

7.1. Bloque PC prinicpal

El bloque PC Principal es el encargado de proveer la próxima dirección de memoria, en base a si hay que realizar algún tipo de salto, a si se debe incrementar el PC o si se debe mantener el valor del PC constante. Al realizar el diseño de este bloque, se tuvieron en cuenta los siguientes items.

- La memoria de programa es una memoria sincrónica (limitación del software de simulación utilizado), por lo que se necesita una señal de clock para actualizar su salida. Es decir, se puede interpretar como que a la salida de la memoria hay un latch asociado a esta.
- Conectado en serie con la salida de dicha memoria, hay un latch (Instruction Register) que contendrá el valor de la instrucción que se está ejecutando. Por lo tanto, hay dos latches en serie (la salida de la memoria, y el Instruction Register), por lo que uno de estos podría ser, en principio, omitido.

- La memoria de microinstrucciones (control store) también es una memoria sincrónica (debido a la misma limitación), y el valor del address debe estar disponible a la entrada de dicha memoria antes de se produzca el clock que actualiza la salida de la memoria de microinstrucciones.

Analizando esto en detalle, se desprende la siguiente conclusión:

- El valor de la instrucción debe estar disponible para la memoria de microinstrucciones antes de que este sea almacenado en el latch (Instruction Regiser). Esto implica que la salida de la memoria de programa debe proveer el valor de la microinstrucción antes de que dicho valor sea guardado por el Instruction Register. Esto significa que el valor de la instrucción debe estar disponible antes del clock que hace que esta sea almacenada en el Instruction Register, es decir, ambos latches son necesarios, y el address de la memoria de microinstrucciones (contenido dentro del valor de la instrucción a ejecutar) es obtenido leyendo la salida de la memoria de programa, y no la salida del Instruction Register.

Considerando que una instrucción debe procesarse en el tiempo que dura un clock, y la nueva instrucción debe encontrarse disponible antes del próximo flanco ascendente, entonces:

- La memoria de programa debe ser leída antes del flanco ascendente, dado que en el flanco ascendente debe almacenarse el valor de la instrucción en el Instruction Register, por lo que en la salida de la memoria debe estar disponible dicho valor. Entonces, se decide realizar la lectura de la memoria en el flanco descendente.
- Dado que la lectura se realiza en el flanco descendente, el valor de dirección de memoria debe estar determinado antes de dicho flanco. Si el bloque PC recibe un flanco ascendente indicando que debe preparar la próxima instrucción, a partir de este momento la dirección debe estar disponible dentro de medio ciclo de clock.

Finalmente se tiene en cuenta que:

- La acción a realizar (esto es, realizar un salto, mantener el valor actual del PC o incrementar el PC) se define en función de la microinstrucción actual. Esta microinstrucción está disponible a partir de un flanco ascendente, y debido a retardos de propagación se pueden producir glitches en las entradas de control del PC, que pueden indicar de manera errónea que hay que realizar determinada acción. Estas señales se estabilizan luego de un tiempo (menor a medio ciclo de clock). El bloque PC debe ser inmune a estos glitches, y la salida ser la correcta pasado medio ciclo de clock.

Teniendo todos estos factores en cuenta, se realizó el circuito de la Figura 4

Este circuito, dada la dirección de memoria actual, la cual se mantiene una copia en el latch interno (siempre disponible en la salida $Curr\_addr[10..0]$), puede ofrecer 3 salidas distintas en función de las señales de control $hold\,(h)$ y $load\,(l)$.

- Si $h = 1$, $Inst\_addr[10..0]$ entrega el valor almacenado en el latch interno. Esto significa que la dirección de la próxima instrucción a ejecutar es la misma que la dirección actual, por lo que el valor del PC no se incrementa
- Si $h = 0$ y $l = 0$, $Inst\_addr[10..0]$ entrega el valor almacenado en el latch interno, incrementado en uno. Esto significa que la dirección de la próxima instrucción a ejecutar es la siguiente a la de la instrucción anterior, es decir, se incrementa el PC
- Si $h = 0$ y $l = 1$, $Inst\_addr[10..0]$ entrega el determinado por la entrada $Jmp\_addr[10..0]$, por lo que se produce un salto.

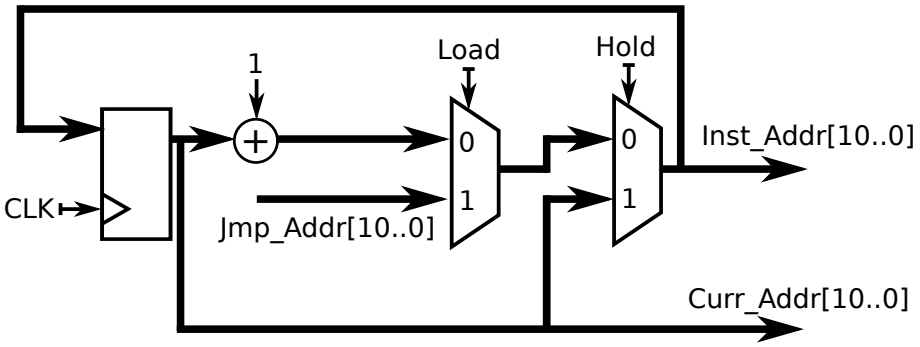

Figura 4: Bloque PC principal

Es importante resaltar que las señales *load* y *hold* controlan las entradas de los muxes, que son subcircuitos combinacionales, por lo que no es un problema si se presentan glitches en dichas señales, siempre y cuando la señal se estabilice antes de la lectura de memoria, que ocurre en el flanco descendente. Entonces

- Antes de un flanco ascendente, la salida del bloque PC representa la dirección de la próxima instrucción a ejecutar, y a la entrada del Instruction Register se encuentra el valor de la próxima instrucción a ejecutar (estado inicial del circuto). Cabe aclarar también que, en todo momento, el valor del latch interno está sincronizado con el valor del Instrucion Register, es decir, en todo momento el latch interno contiene la dirección de memoria donde se encuentra almacenada la instrucción que se encuentra guardada en el Instruction Register (el funcionamiento del bloque PC garantizará esto).
- Ante el flanco ascendente, se comienza a ejecutar dicha instrucción, y en el mismo instante el bloque PC guarda la dirección de memoria asociada a dicha instrucción en su latch interno. Al comenzar a ejecutarse la instrucción correspondiente, se propagan señales, generando posibles glitches en *load* y *hold* (principalmente debido a los delays de los bloques que detectan dependencias), pero dichas señales se estabilizan antes del flanco descendente, quedando disponible a la salida del bloque PC la dirección de la próxima instrucción a ejecutar
- Al llegar el flanco descendente, la memoria de programa utiliza dicha dirección y pone en la salida de la memoria la instrucción a ejecutar. Entonces, la salida del bloque PC representa la dirección de la próxima instrucción a ejecutar, y a la entrada del Instruction Register se encuentra el valor de la próxima instrucción a ejecutar (estado final del circuto). A partir de este punto, se repite el proceso.

Este diseño permite cumplir con los requisitos que se determinaron anteriormente. Debido al funcionamiento del circuito, el valor inicial del PC debe tener inicialmente todos sus bits en 1, para que la primera instrucción leída sea la que se encuentra en el address 0x000000. Este circuito no se encarga de determinar cuándo debe realizarse un salto, incrementar el PC o mantener su valor, por lo que esto es determinado por un circuito externo. Adicionalmente, este bloque únicamente procesa saltos, por lo que para implementar la posibilidad de ejecutar subrutinas y regresar a una dirección determinada, hay que agregar otro bloque que se encargue de ofrecerle al bloque PC las señales necesarias.

7.2. Dependencias de salto

Las dependencias de salto son determinadas por el bloque UC-2, entregando las salidas *hold* y *hold_jump*. La entrada *hold* del bloque PC es, en realidad, un OR lógico entre las entradas *hold* y *hold_jump* generadas por el bloque UC-2. Entonces, ante una dependencia, el bloque PC simplemente mantendrá su valor actual, sin importar el valor de *load* entregado por el bloque de condición de salto.

A modo de ejemplo, hay dependencia de salto en una situación como la siguiente:

-Un salto si W es negativo (*JNE*) luego de una instrucción que escribe en W. En este caso, el procesador debe mantener en la primera etapa del pipeline el salto hasta que se termina de ejecutar la anterior instrucción.

Este circuito no forma parte del bloque PC.

7.3. Bloque de condición de salto

El bloque de condición de saltos determina si un salto debe ser tomado o no, ignorando cualquier dependencia en el pipeline. Esto significa:

- Si la instrucción es *JMP X*, *BRA S*, *RET*, el bloque indica que el salto debe ser tomado.
- Si la instrucción es *JCY*, el bloque informa que el salto debe ser tomado si $Cy = 1$
- Si la instrucción es *JZE*, el bloque informa que el salto debe ser tomado si $W = 0$
- Si la instrucción es *JNE*, el bloque informa que el salto debe ser tomado si W es un valor negativo, si es interpretado como un número signado.

Este bloque genera la salida *load* que se conecta a la entrada del bloque principal de PC. Es importante recordar que las dependencias son determinadas por el bloque UC-2, y que dicha información es recibida por el bloque PC principal y, en caso de dependencias, ignorará las solicitudes de salto del bloque de condición de salto.

7.4. Bloque para el control de saltos a subrutinas

Para implementar los saltos a subrutinas y la lógica para regresar de una subrutina, es necesario implementar un bloque que se encargue de mantener un stack de direcciones. Este bloque se muestra en la Figura 5. Visto con un alto nivel de abstracción, el bloque de control de subrutinas es simplemente un bloque que se conecta a la entrada *Jmp_addr*[10..0] del bloque PC principal. Este bloque determinará la dirección del salto a ejecutar, la cual estará dada directamente por la instrucción actual en caso de una instrucción de salto común, será calculada a partir del valor actual del PC (recibido en la señal *Curr_addr*[10..0], obtenida como salida del bloque PC) en el caso de la instrucción *BRA S*, o entregará una dirección de memoria almacenada internamente indicando la dirección de la memoria que contiene la instrucción que debe ejecutarse para que se "regrese" de una subrutina en el caso de la instrucción *RET*. Es decir, este bloque es simplemente un bloque que determina una dirección de memoria en base al contexto.

Las señales de control de este bloque son *BRA* y *RET*, que indican si la instrucción actual es un branch o un return. Estas señales son obtenidas a partir del análisis del valor de la instrucción actual. Adicionalmente, se tiene el valor de la dirección de memoria de la instrucción actual en *Curr_addr*[10..0], que permitirá calcular la dirección asociada a un salto relativo, y almacenar dicha dirección para poder ser utilizada ante una instrucción *RET*, y también recibe *hold*, determinado de la misma manera que la

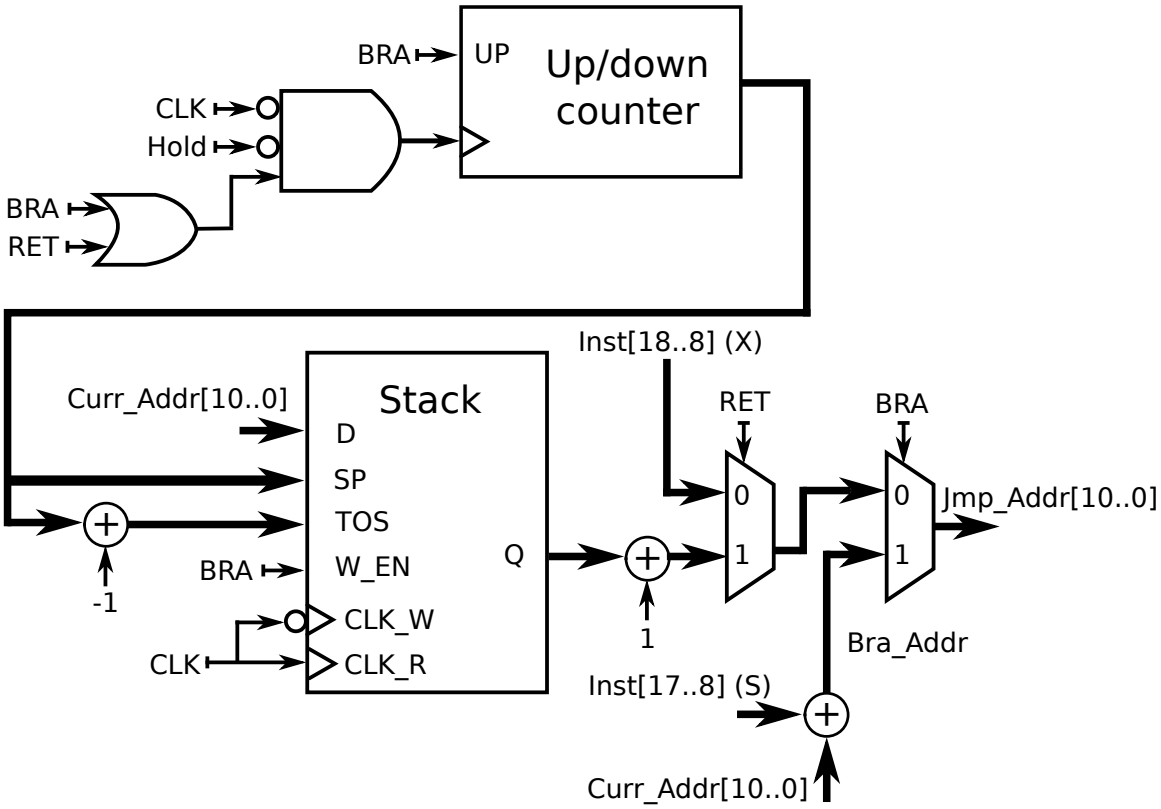

Figura 5: Bloque para control de saltos a subrutinas

entrada *hold* del bloque PC principal. Adicionalmente, se recibe el clock. Es necesario destacar que la dirección de memoria de la próxima instrucción a ejecutar debe ser determinada en menos de medio ciclo (contado a partir del flanco ascendente), para que el bloque PC principal pueda ejecutarla y la dirección correcta esté disponible a la entrada de la memoria de programa antes del flanco descendente del clock.

Este circuito se puede separar en varios subcircuitos

- Subcircuito MUX: Una serie de MUX determina la dirección ofrecida a la salida del bloque.
  - Si $RET = 0$ y $BRA = 0$, se asume que la dirección será un salto y se ofrece la dirección de salto a la salida, obtenida directamente a partir de parte del valor de la instrucción actual ($Addr = Inst[18..8]$).
  - Si $RET = 0$ y $BRA = 1$, se calcula la dirección del salto a partir de la dirección actual ($Curr\_addr[10..0]$), al que se le suma el offset indicado por $S = Inst[17..8]$. Esta suma se hace con extensión de signo y es una suma signada. Este valor es calculado con un circuito combinacional.
  - $RET = 1$ y $BRA = 0$, La salida está dada por el TOP OF STACK de la memoria stack, incrementado en 1. Esto representará la dirección de regreso"de una subrutina. Este valor es calculado con un circuito secuencial, por lo que hay que garantizar que dicho valor esté disponible a la salida antes de medio ciclo de clock.

  - la condición $RET = 1$ y $BRA = 1$ no se puede dar.

- Subcircuito de punteros: Este subcircuito es un up-down counter que generará el valor del puntero STACK POINTER (SP), y a partir de este valor se calcula el puntero TOP OF STACK (TOS=SP-1). Estos punteros se utilizarán para leer y escribir la memoria de stack. El valor del contador cambiará siempre que se trate de una instrucción BRA o RET, únicamente si la entrada $hold$ se encuentra en 0, y el cambio del valor ocurrirá en el flanco descendente del clock. La señal de hold se utiliza para que, ante posibles dependencias del pipeline (debidas quizá a operaciones no relacionadas con el salto), los valores de los punteros no se modifiquen. Notar que los posibles glitches en la señal de $hold$ ocurren cuando $CLK = 1$, quedando entonces enmascarados (ver circuito de Figura 5).
- Memoria de Stack: Es la memoria utilizada para guardar las direcciones ante un BRA y obteneras en caso de un RET.
  - Ante un BRA, se guarda la dirección de memoria actual (obtenida externamente). La operación de escritura se realiza en el flanco descendente, utilizando el valor del SP antes de que este sea incrementado (de no haber dependencias).
  - Ante un RET, se lee el valor apuntado por TOP OF STACK. Esta operación de lectura se produce en un flanco ascendente, para que durante la ejecución de la instrucción (y, en particular, durante la primera mitad del clock) la salida de la memoria sea la dirección correspondiente a la del último BRA, y que la salida de la memoria quede actualizada para la próxima instrucción al finalizar la operación actual (permitiendo que dos RET seguidos funcionen de forma correcta). El valor de la salida de la memoria de stack es incrementado en 1 antes de ingresar al subcircuito MUX.

La combinación de los circuitos PC Principal, bloque de condición de salto y bloque para control de saltos a subrutinas componen el bloque PC representado en la Figura 1.

8. Ampliación de ISA: direccionamiento indexado

8.1. Motivación

El procesador EV20, en su versión original, tiene una característica que hace que una gran variedad de programas sean muy engorrosos de escribir y ocupen mucha memoria: sólo puede accederse a la memoria de datos en con modo de direccionamiento directo. Esto quiere decir que el programa debe tener, al menos, una instrucción que explicite cada dirección a la que se quiere acceder.

Esto resulta engorroso para trabajar con estructuras como strings y arreglos, donde en general al programador le basta definir la posición del primer elemento, la cantidad de elementos y el tamaño de cada uno. Tener estos tres datos explícitos en el programa resulta mucho más natural y sencillo que tener que explicitar la dirección de cada elemento, que sería además mucho más susceptible a errores.

Sin embargo, el procesador utilizado tiene otra característica que hace que añadir un modo de direccionamiento más cómodo sea relativamente sencillo. Como el bus de direcciones de memoria tiene menos bits (11) que el bus de datos (16), en principio cualquier registro podría utilizarse para guardar una posición de memoria (ya que el tamaño de los registros es en este caso el mismo que el de los datos).

Se procedió, pues, a implementar modo de direccionamiento indexado en el EV20. Se tomó la decisión de dar la opción de usar dos registros para permitir más flexibilidad (por ejemplo, se podrían hacer operaciones del tipo *string copy*, o calcular la diferencia entre dos posiciones para determinar la cantidad

de elementos en el medio), dado que la complejidad no aumentaba demasiado entre un registro y dos. Por simplicidad se designaron a los registros R_0 y R_1 para este propósito.

Estos registros son de 16 bits, con lo cual se ignorarán los 4 bits más significativos a la hora de utilizarlo para acceder a la memoria.

8.2. Modificaciones del set de instrucciones

Se procuró mantener el set de instrucciones lo más parecido posible al original. Sin embargo, no introducir ninguna modificación en él habría implicado que ya no fuera suficiente mirar a los primeros 7 bits de la instrucción para determinar su tipo, lo cual resultaría a su vez en un cambio sustancial en la decodificación.

Por otro lado, existen ya en el EV20 dos instrucciones que acceden a memoria: *MOM Y, W* y *MOM W,Y* (*move operand to memory* y *move operand from memory* respectivamente). Que los accesos a memoria se procesen de forma fundamentalmente diferente para modo indexado y para modo directo también agregaría complejidad. Por lo tanto, se decidió que se modificarían estas dos instrucciones de manera tal que sólo un bit diferencie el modo de direccionamiento.

La estructura original de estas instrucciones está representada en la tabla 4.

Instrucción	ID	Dirección	Don't care
MOM W,Y	0101	yyy yyyy yyyy	dddd dddd
MOM Y,W	0100	yyy yyyy yyyy	dddd dddd

Tabla 4: Instrucciones de lectura y escritura del EV20 original

Los últimos ocho bits no son tomados en cuenta por el EV20 salvo para las instrucciones que trabajan con constantes. Se decidió, pues, avanzar sobre estos bits para modificar las instrucciones.

Estas dos instrucciones se convirtieron pues en las cuatro que se observan en la tabla 5.

Instrucción	ID	*i*	Dirección	Don't care
MOM W,Y	01010	d	yyy yyyy yyyy	dd dddd
MOM Y,W	01000	d	yyy yyyy yyyy	dd dddd
MOM W,Y $+ R_i$	01011	i	yyy yyyy yyyy	dd dddd
MOM Y $+ R_i$*,W*	01001	i	yyy yyyy yyyy	dd dddd

Tabla 5: Instrucciones de lectura y escritura del EV20 con modo de direccionamiento

Comparando las tablas 4 y 5 puede observarse que los primeros 4 bits del ID se mantuvieron iguales, agregando un bit que es 0 para las instrucciones de modo directo y 1 para las de indexado. Efectivamente, podría interpretarse como que estas cuatro instrucciones son en realidad dos: los primeros 4 bits indican que se trata de una operación de escritura o de lectura a memoria, el siguiente el modo de direccionamiento, seguido por un bit que indica si se usa el registro 0 o el 1 para el indexado (que es don't care para modo directo), y por último la dirección a la que se referencia. Si bien para el caso indexado, Y sería el offset en lugar de la dirección, en la práctica esta distinción no hace diferencia, puesto que la dirección $Y + R_i$ será la misma en ambos casos.

8.3. Modificaciones de hardware

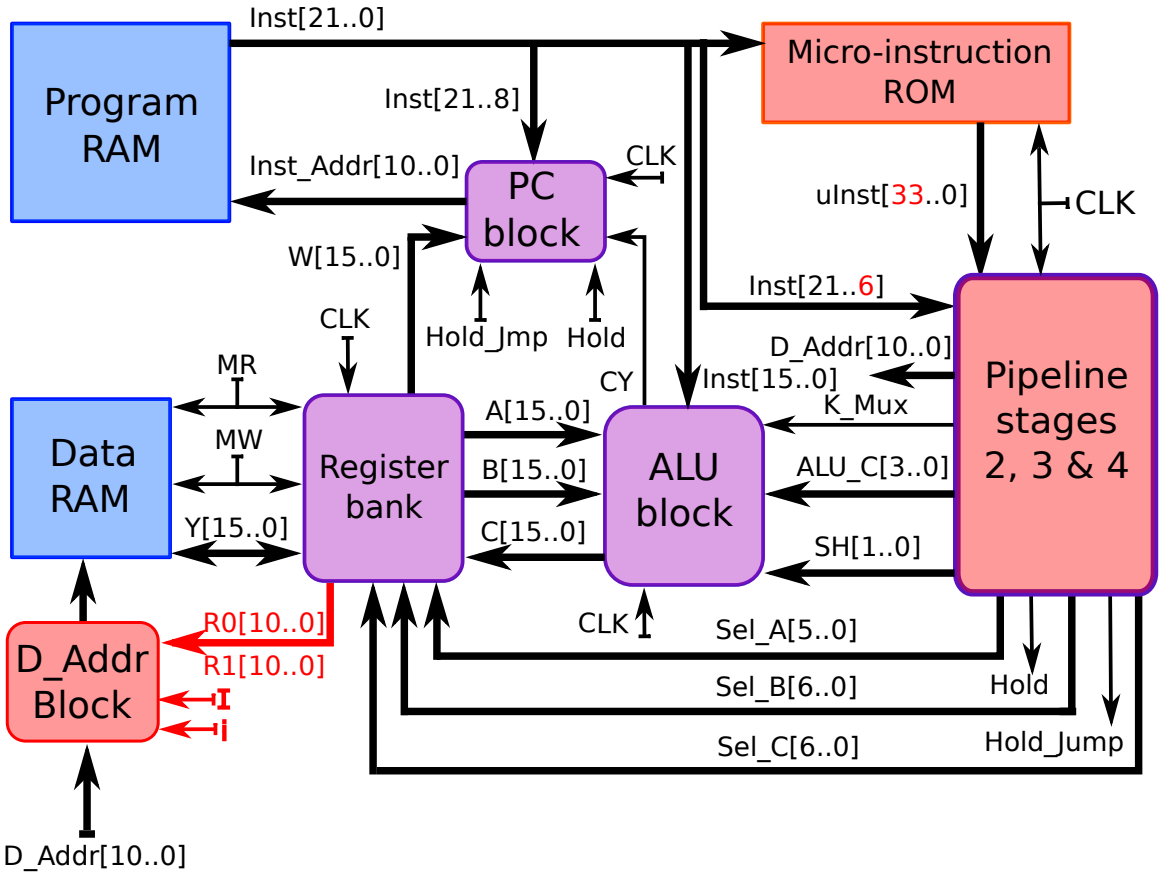

Figura 6: Diagrama de bloques del EV20 con modo de direccionamiento indexado

Las modificaciones realizadas a nivel hardware se pueden observar en la figura 6. Concretamente son:

- modificaciones de la *micro-instruction ROM*: se agregaron dos bits de microinstrucción
  - I: es 1 para acceso de memoria en modo indexado, 0 caso contrario
  - i: indica el registro que se utiliza (R_0 o R_1), *don't care* para modo directo
  - I e i se agregaron al campo M de la microinstrucción, que pasó de tener 2 a 4 bits
  - para las instrucciones que no acceden a memoria, $i = I = 0$, pero pueden considerarse *don't cares*, ya que como $MR = MW = 0$, no se generará una operación de lectura o escritura de la memoria de todas maneras
- se agregó un bloque encargado de obtener la dirección de la memoria a la que se quiere acceder, a partir de D_Addr, R_0, R_1, I e i. Consiste en:
  - dos sumadores para obtener las direcciones D_Addr+R_0 y D_Addr+R_1
  - dos multiplexores:

    - uno que elige entre *D_Addr*+R_0 y *D_Addr*+R_1, utilizando el bit *i* de la microinstrucción como línea de selección
    - uno que elige entre D_Addr y la salida del primer multiplexor, utilizando el bit I de la microinstrucción como línea de selección

- modificaciones del bloque pipeline
    - *D_Addr* se obtiene de *Inst[15..6]* en lugar de *Inst[17..8]*
    - el sub-bloque UC1 intercepta los bits *i* e I, ya que recibe la totalidad del bloque *M* para no generar operaciones de lectura/escritura cuando el pipeline debe frenarse
    - el sub-bloque UC2 recibe los bits I e *i* de la etapa 2 (es decir, continúa recibiendo la totalidad de *M*) para poder determinar dependencias RAW de R_0 y R_1, para el caso particular que la lectura de estos registros venga dada por una instrucción de acceso a memoria en modo indexado

9. Validación

Para comprobar el correcto funcionamiento del procesador, se hicieron simulaciones sobre distintos módulos y simulaciones con distintos programas.

9.1. Entorno de simulación y configuración

Se escogió utilizar el entorno de simulación de modelSim principalmente por los siguientes motivos:

- Viene incluido dentro del paquete de Quartus.
- El tiempo de compilación y de simulación es significativamente menor que las simulaciones propias de Quartus.
- Permite detallar por código entradas y configuraciones de simulación. Por ejemplo, se pueden escribir que se prueben todos los comandos de ALU.

9.2. Pruebas modulares

Antes de realizar pruebas con todo el procesador conectado, se probaron distintos módulos.

Primero se escribió una simulación sobre el conjunto Latches-ALU-Shift, en la que se probaban todos los códigos de control. Se verificó manualmente que las salidas sean las esperadas, incluyendo la salida del registro C y la señal de carry.

Luego se probó la lectura y escritura sobre el banco de registros, considerando los casos de memoria.

Por último, se probó el flujo de las micro instrucciones dentro de las etapas 2, 3 y 4, sin considerar problemas de dependencia.

Al funcionar todos estos bloques correctamente, se procedió a probar el conjunto.

9.3. Programas probados

Se diseñaron programas para pruebas específicas de funcionamiento, escribiendo manualmente el contenido de la memoria del programa.

9.3.1. Dependencias RAW en registro W

Se desarrolló un simple programa con carga de valores predeterminados y operaciones simples sobre el registro de trabajo. Con ella se pudo comprobar el funcionamiento del bloque de control de pipeline para resolver dependencias RAW en el registro de trabajo.

9.3.2. Escritura y lectura a memoria

Se diseño un programa en el cuál se hacían lecturas y escrituras a memoria. En el mismo se validó que se escribía correctamente el valor programado en la memoria deseada, así como su lectura. También se descubrió en este programa la dependencia WAW por tener la operación de traer de memoria menos etapas de pipeline.

9.3.3. Utilización de todas las instrucciones exceptuando las de salto y subrutinas

Con este código se verificó el correcto funcionamiento de las instrucciones que no fueron probadas en los anteriores programas, sin incluir a las de saltos y saltos a subrutinas. Se comprobó el funcionamiento de las dependencias RAW en registros.

9.3.4. Pruebas de saltos

Para verificar el funcionamiento de los saltos se implementó un programa en el cuál se utilizaban todos los saltos. Se utilizaron otras instrucciones para asegurar que se den las condiciones para los saltos condicionales, probándose también que no se haga el salto si las condiciones no fueran dadas.

9.3.5. Pruebas de subrutinas

Para terminar de comprobar la funcionalidad del ISA básico del ev-20, se escribió un programa con las instrucciones de subrutinas. Se probaron múltiples niveles de subrutina y su correcto retorno al programa principal.

9.3.6. Pruebas de direccionamiento indexado

Se elaboró un programa que utilizaba las funcionalidades agregadas de direccionamiento indexado. En él se pudo comprobar la correcta escritura y lectura de memoria para los distintos registros de indexación.

9.3.7. Programa para cálculo de sucesión de Fibonacci

Se implementó un programa que calcula sucesión de Fibonacci, para probar el correcto funcionamiento del modo de direccionamiento indexado, utilizando tanto el registro R0 como el registro R1. Con el objetivo de mostrar un programa completo, se incluye a continuación el código de este programa (Programa 1).

Programa 1: Fibonacci

```
#ORG 0

MOV W,#0                    -- Cargo 0 a W
MOV R0,W                    -- Lo guardo en R0 (es mi puntero) -> R0-> 0
    ↪ x0000
MOV [R0+0],W                -- Condiciones iniciales fibonacci F0 = 0
MOV W,#1                    -- Cargo 1 a W
```

```
MOV [R0+1],W            -- Condiciones iniciales fibonacci F1 = 1
MOV W,#10               -- Iteraciones de fibonacci a realizar (-1)
MOV R2,W                -- R2 contiene contador

LOOP1:
MOV W,[R0+0]            -- Cargo Fn-2 en R3
MOV R3,W
MOV W,[R0+1]            -- Cargo Fn-1
CLR CY                  -- Borro Carry
ADR W,R3                -- W = W + R3 + Cy --> W = Fn-2 + Fn-1
MOV [R0+2],W            -- Guardo Fn
MOV W,R0                -- Preparo para incrementar puntero
CLR CY                  -- Borro Carry
ADK W,#1                -- W = W+1+Cy
MOV R0,W                -- Termino de incrementar puntero
MOV W,R2                -- Rescato Contador
CLR CY                  -- Borro Carry
ADK W,#-1               -- Decremento Contador
MOV R2,W                -- guardo contador
JCY LOOP1               -- Repetir mientras no termine

-- Seguir calculando fibonacci pero con indexado en R1 para probar los
   ↪ dos modos de direccionamiento indirec
MOV W,R0                -- Copiar R0
MOV R1,W                -- A R1 (este es mi nuevo puntero)
MOV W,#10               -- Iteraciones de fibonacci a realizar (-1)
MOV R2,W                -- R2 contiene contador

LOOP2:
MOV W,[R1+0]            -- Cargo Fn-2 en R3
MOV R3,W
MOV W,[R1+1]            -- Cargo Fn-1
CLR CY                  -- Borro Carry
ADR W,R3                -- W = W + R3 + Cy --> W = Fn-2 + Fn-1
MOV [R1+2],W            -- Guardo Fn
MOV W,R1                -- Preparo para incrementar puntero
CLR CY                  -- Borro Carry
ADK W,#1                -- W = W+1+Cy
MOV R1,W                -- Termino de incrementar puntero
MOV W,R2                -- Rescato Contador
CLR CY                  -- Borro Carry
ADK W,#-1               -- Decremento Contador
MOV R2,W                -- guardo contador
JCY LOOP2               -- Repetir mientras no termine

END:
JMP END                 -- Fin de programa
```

El resultado de este programa se representa en la Tabla 6

Dirección de Memoria	Valor	Dirección de Memoria	Valor
0x00000	0	0x0000C	144
0x00001	1	0x0000D	233
0x00002	1	0x0000E	377
0x00003	2	0x0000F	610
0x00004	3	0x00010	987
0x00005	5	0x00011	1597
0x00006	8	0x00012	2584
0x00007	13	0x00013	4148
0x00008	21	0x00014	6765
0x00009	34	0x00015	10946
0x0000A	55	0x00016	17711
0x0000B	89	0x00017	28657

Tabla 6: Memory Dump obtenido a partir de la ejecución del programa. Representación decimal no signada

10. Conclusiones

Se implementó un procesador EV20 de arquitectura Harvard con una pipeline de 5 etapas, con especial atención a que la misma sólo se detuviera de ser absolutamente necesario para maximizar la performance.

Este procesador provino de adaptar el EV16, siendo la principal diferencia que utiliza registros y datos de 16 bits, lo cual requirió aumentar el tamaño de las microinstrucciones en 8 bits.

Como propuesta de mejora, se crearon dos instrucciones que utilizan modo de direccionamiento indexado para acceder a memoria, lo cual implico agregar dos bits más a la microinstrucciones de acceso a memoria (uno indicando modo de direccionamiento, y otro indicando cuál de los dos registros posibles se debe usar como dirección base). El hardware extra para implementar esta funcionalidad se redujo a pasar el bus D_Addr por dos multiplexores: uno que determinara el registro de indexado, y luego otro para seleccionar el modo de direccionamiento. Además, para obtener los valores $Y + R_0$ y $Y + R_1$ se incorporaron dos sumadores.

En cuanto al set de instrucciones, sólo se modificaron dos de las existentes instrucciones (justamente, las de acceso a memoria), y no se agregaron bits a la instrucción.

Esta pequeña mejora (en cuanto a modificaciones requeridas) permitió agregar mucha funcionalidad al procesador, sobre todo para el procesamiento de arreglos y strings. Esto se vio demostrado en la implementación de un código escrito en assembler que permitiía calcular los números de Fibonacci y guardarlos en memoria, lo cual si bien es un objetivo humilde, hubiese sido mucho más engorroso de implementar si se tuviese que haber explicitado cada dirección de memoria, ya que no se podría haber hecho ningún *loop* para resolver el problema.