CS2201 - Computer Architecture: 2020-1 Prof.Jorge Gonzalez

**Final project - Microprocessor**

**Integrantes:**

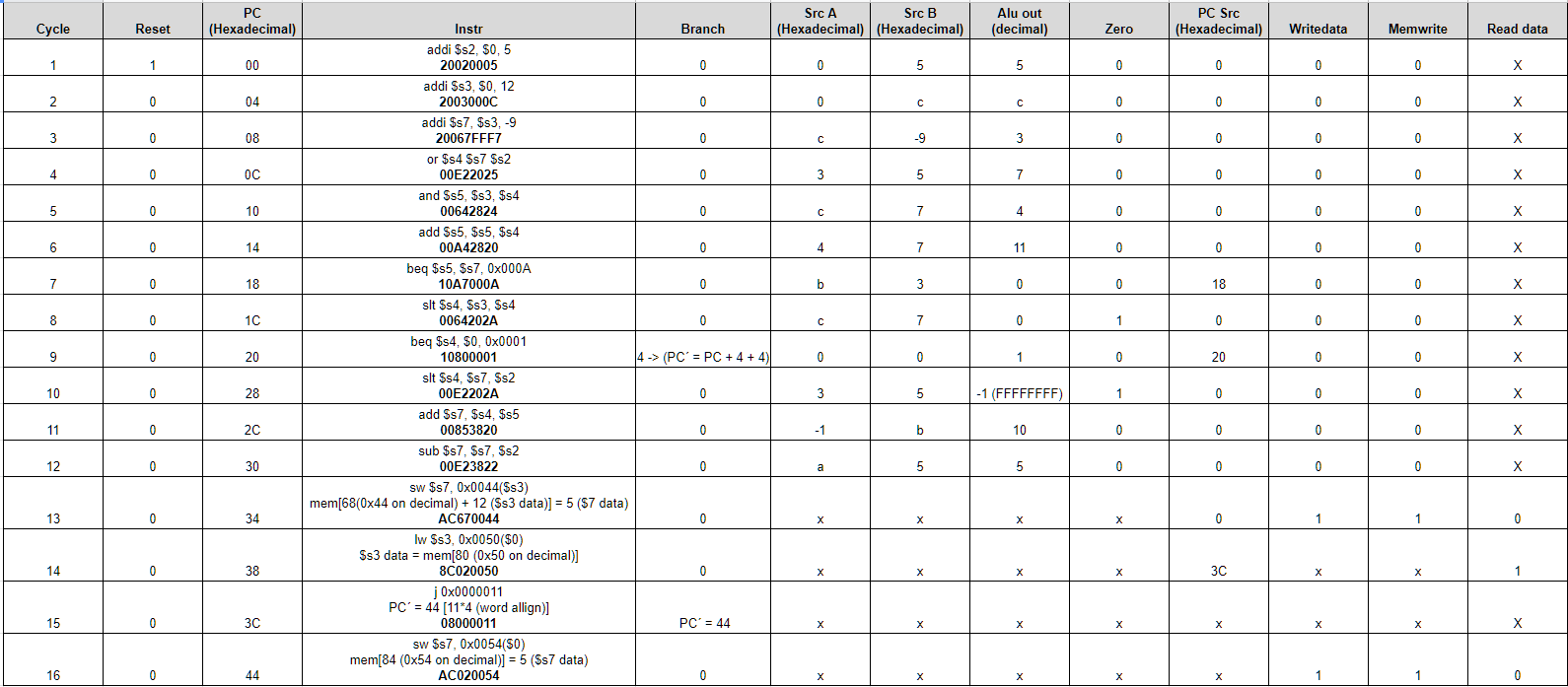
* Eduardo Sebastián Castro Quispe 201910111
* Efraín Córdova Amaya 201910129

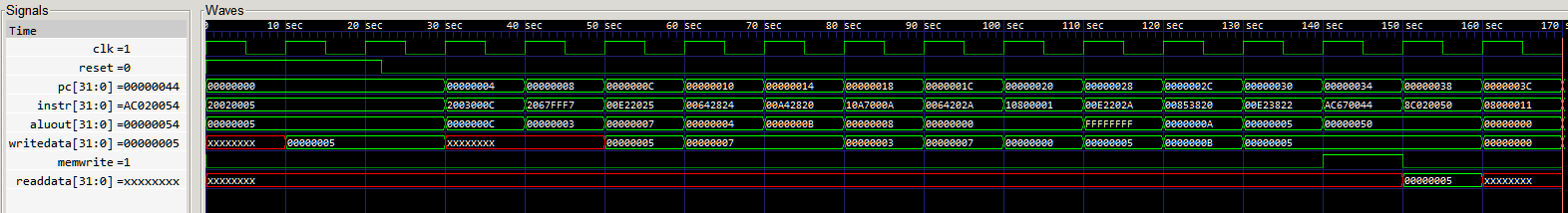
**Lab 7: Single - Cycle Processor**

* Descripción de la tarea: Para este laboratorio, se nos pide conectar nuestro ALU del laboratorio 5 a un single-cycle processor. Luego, se debe crear un programa que testee el funcionamiento de este a través de una secuencia de instrucciones. Finalmente, se implementarán dos nuevas operaciones al microprocesador y se evaluarán con un segundo conjunto de instrucciones.

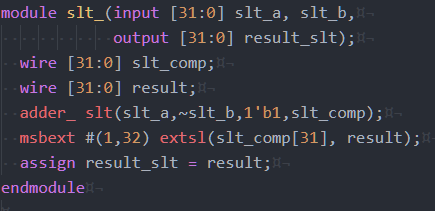
**1 - Table 1 - Understanding instructions:**

En los archivos del informe puede encontrar un .xlsx con la tabla en excel.

****

**2- Simulación del programa**

Una vez conectamos nuestro ALU al procesador e identificamos posibles errores de implementación, obtenemos las siguientes señales en nuestro programa. Rápidamente, y tal como lo notamos al realizar la tabla 1, obtenemos un primer “Simulation failed” por parte del sistema. Ello se debe al valor final de writedata, el cual no coincide con el valor esperado (7). ¿Cómo explicamos esto? Recordemos, por un momento, el funcionamiento de la operación slt de nuestro ALU:

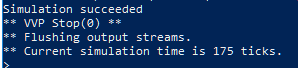


Si nos fijamos en las operaciones, vemos que el output es de 32 bits, como debe ser todo resultado del ALU. Sin embargo, ¿cómo representamos 1-lógico en 32 bits? Para este slt, realizamos un sign (msb) extend. Ello implica que el valor que obtendremos en caso de cumplirse la condición de a < b es **32’hFFFFFFFF** y no **32’h00000001.** Recordemos que en nuestro procesador el trigésimo primer bit de un resultado es considerado como el signo negativo y que, además, usamos la regla de complemento 2 para este tipo de valores. En otras palabras, para operaciones del ALU, slt es -1 cuando la condición se cumple y 0 cuando no. Por ello, el resultado del addi $s7, $s4, $s5 es 10 (b-1) y no 12 (b+1). Esta política de implementación con el slt nos lleva a concluir que el valor que espera el programa se encuentra dos unidades por encima del real. Con el fin de mantener nuestro ALU como en anteriores labs, nos limitaremos a cambiar la condición que aprueba la funcionalidad del programa de writedata === 7 a **writedata === 5.** En los siguientes apartados de esta parte del informe, tendremos que definir otra política de extensión de bits menores a 32, y, para ese casos, resultará más interesante que hacer cambiar un condicional.

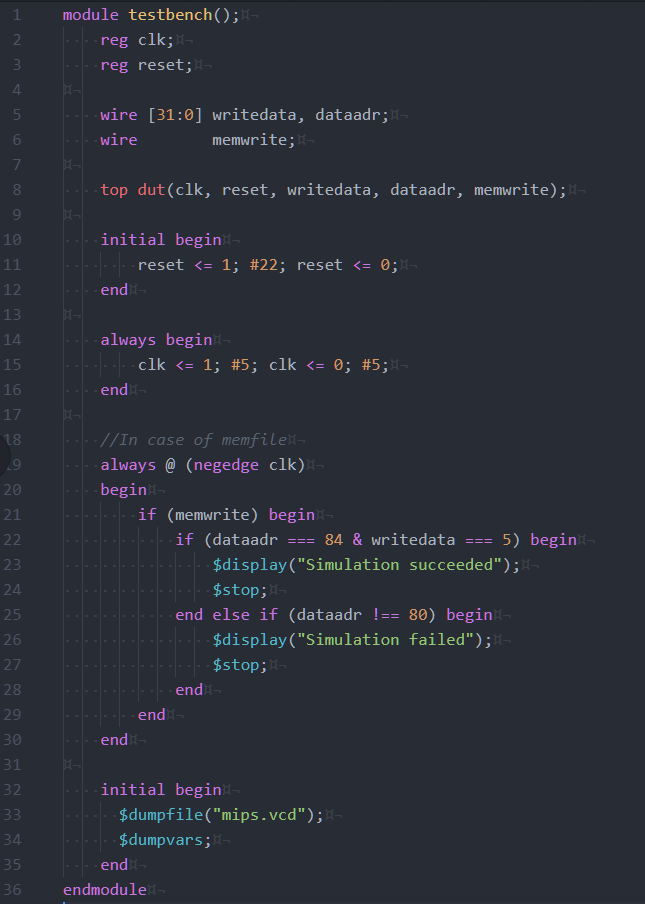
Entonces, cambiamos el módulo de forma que el condicional se active cuando writedata = 5



Finalmente, y luego de cambiar el condicional, obtenemos un “Simulation succeeded” por parte del terminal.

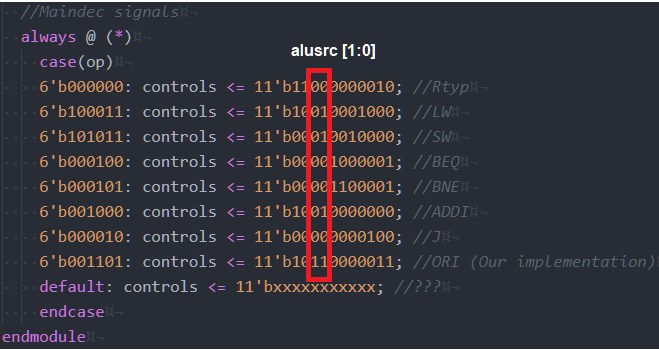
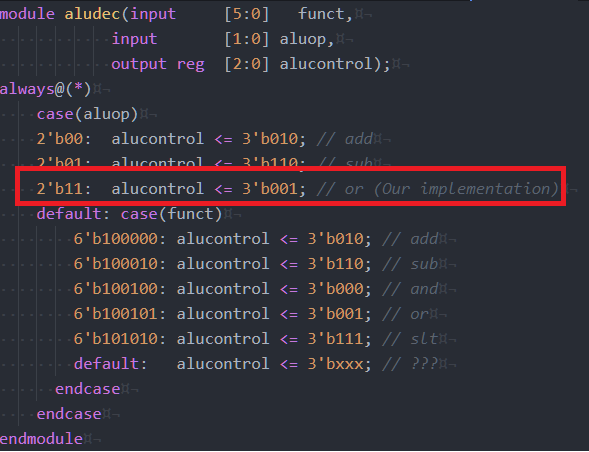
****

Recordar que el módulo testbench final sería:

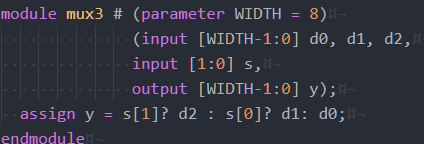
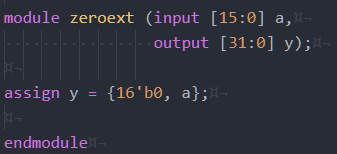


**3 & 4 - Nuevas instrucciones:**

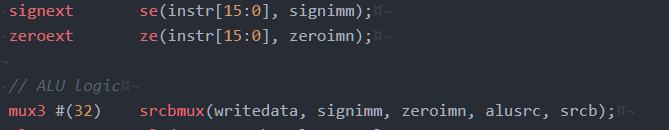
ORI: Para esta instrucción, debemos implementar una lógica que nos permita realizar la operación lógica OR utilizando el aluop, de forma que podamos definirla en el main\_decoder.v. Por otra parte, necesitamos implementar un zero-extend de 16 a 32 bits para el inmidiate que contiene instr[15:0]. Para el slt aún era debatible si aplicar un extend del msb era o no una política acertada, pero para esta implementación es necesario agregar un zero-extend. Adicionalmente, debemos cambiar la lógica del alusrc, ya que ahora será un selector de 2 bits, no de únicamente 1 bit, y cambiar srcbmux para que sea un mux3, no un mux2. Lo siguiente es implementar la lógica que seleccionará al zero-extend de instr[15:0] cuando alusrc coincida con el patrón que se decodifica cuando la operación es un ORI. Esto implica cambiar muy ligeramente la lógica del controller.



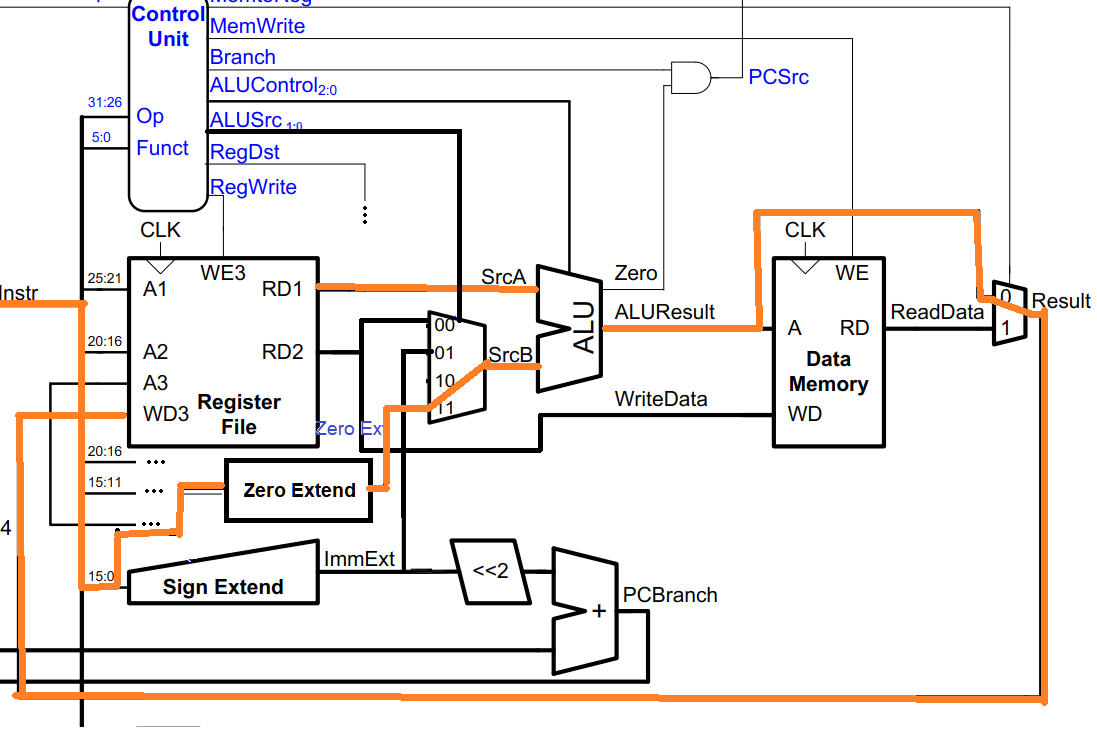
Se implementan estos dos módulos:

****

Luego, adaptamos e implementamos los bloques en structural que los vayan a utilizar.

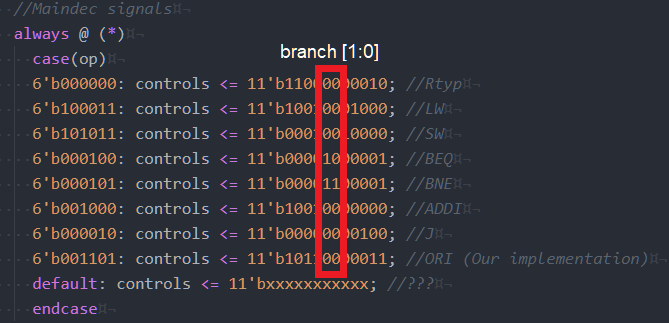
****

El datapath para esta instrucción, sin contar la lógica del próximo pc, que sigue exactamente igual, es:

****

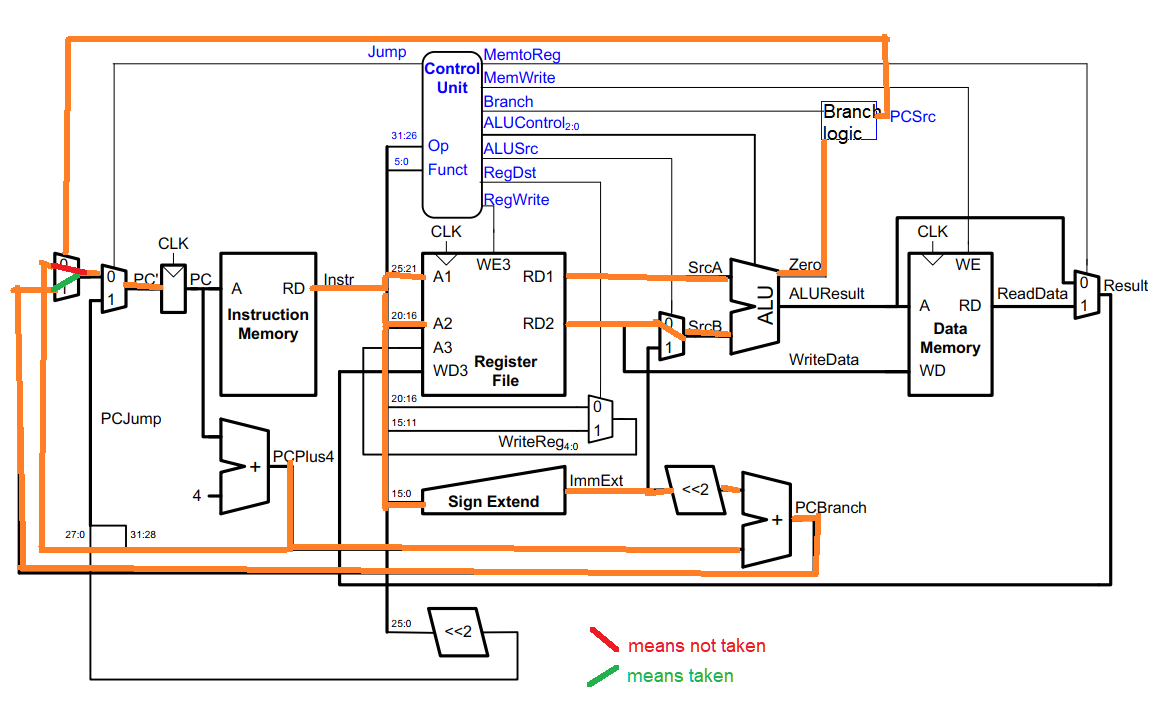
BNE: La lógica de implementación es similar a la realizada en ORI. Primero, debemos adaptar el branch para que sea un selector de dos bits y no únicamente de uno (esto nos permite elegir un resultado de entre 4 opciones como máximo), de forma analóga a alusrc en el ejercicio anterior. Luego, debemos hacer que durante la operación BNE el aluout sea de 01, para restar los operandos, al igual que en BEQ. A su vez, debemos implementar el branch de forma que su valor sea 11 para que la condición de comparación sea la de desigualdad. A diferencia de algunas señales, branch es un cable interno del módulo controller y es allí donde debemos limitar sus acciones e implementar los cambios referente a este. Anteriormente, teniamos que el pcsrc era equivalente a branch & zero. Esto no puede seguir siendo así, ya que branch tiene más posibles valores que sólo 0-lógico y 1-lógico. La idea es usar los bits de branch somo selectores y al literal zero y su complemento como operandos variables según esta señal. De esa forma, tendríamos implementada esta operación.



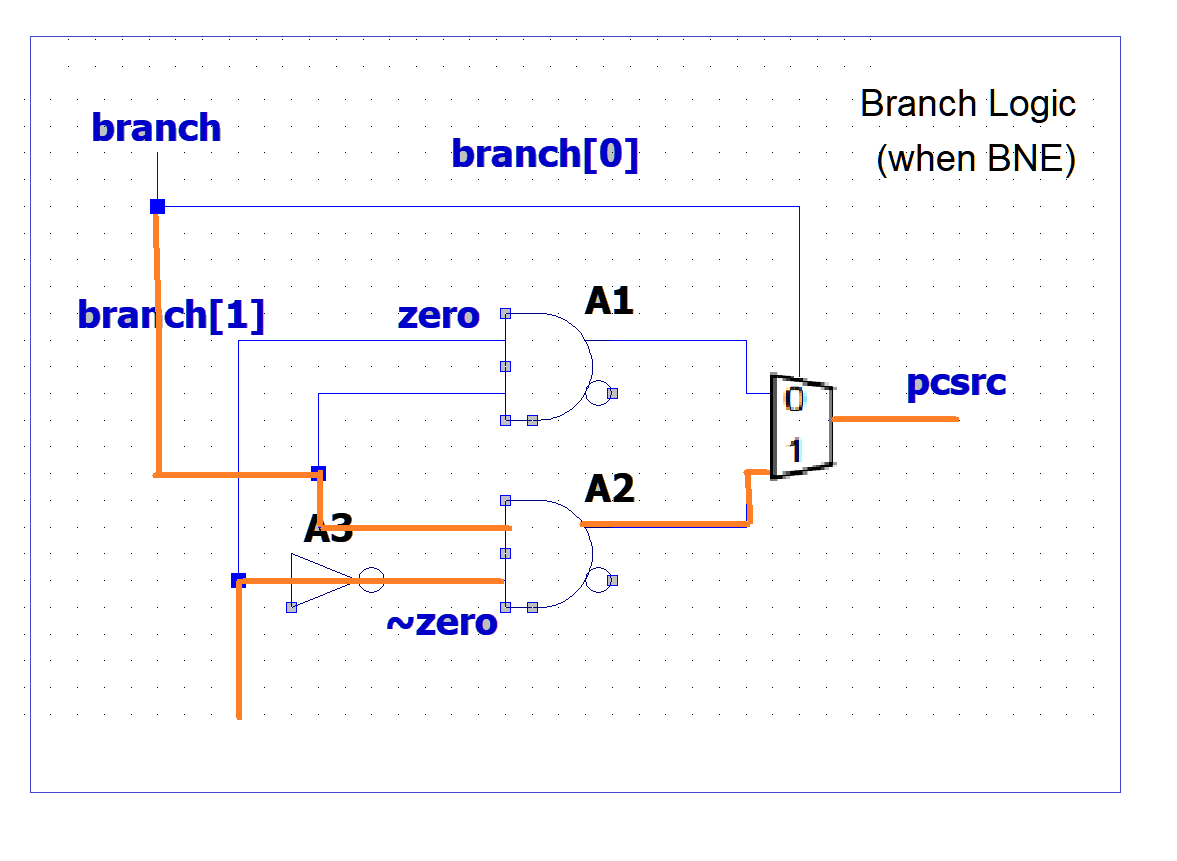
****

Para concluir con BNE, se tiene el datapath de la instrucción:

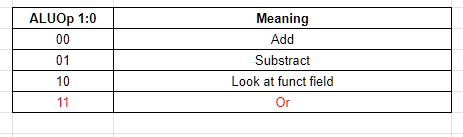
(Nota: Este datapath cumple para branch taken y not taken, ya que ello depende del valor de pcsrc)

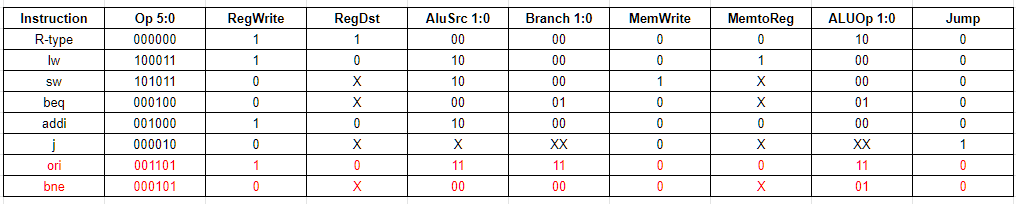
****

El circuito combinacional branch logic sería:



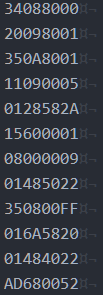
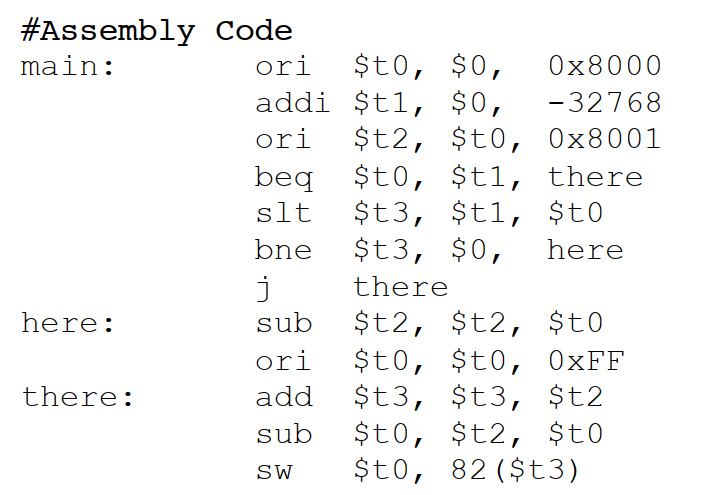
Con nuestras instrucciones ya implementadas se nos pide completar las siguientes tablas:





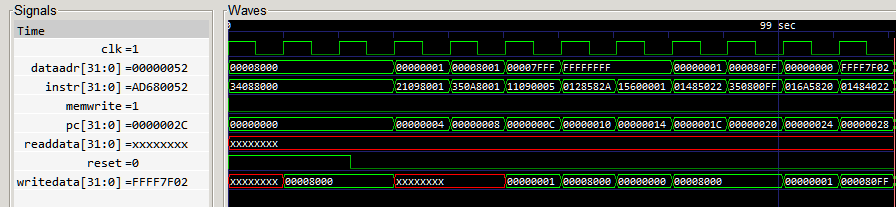
Y con ello, finalmente, podemos pasar al testeo de las instrucciones añadidas.

5 - Transcribir las instrucciones a hexadecimal:



(Nota: La segunda instrucción, addi $t1, $0, -32768; asume que -32768 = 15´h8001, ya que -32768 = -2^15)

**6 - Simulación de las nuevas instrucciones:**

****

Adicionalmente, se ha modificado el testbench para que arroje “Simulation succeeded” si el valor final es 15´hFFFF7F02.

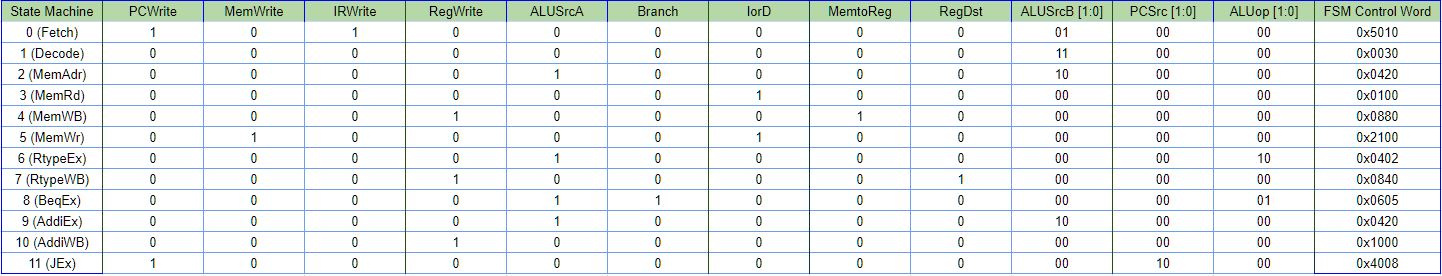
Finalmente, se tiene el módulo testbench editado para estas nuevas instrucciones. Con esto último, hemos terminado la parte de Single-Cycle del informe.



**Lab 8: Multicycle Processor - Part 1: Controller**

* Descripción de la tarea: En esta parte del MCP, se pide terminar el módulo controller. Ello contempla diseñar casi completamente la FSM del procesador, así como las señales que se activarán según que tipo de instrucción este siendo decodificada. Para concluir, se debe desarrollar un programa de test y comprobar el correcto funcionamiento del módulo y sus bloques subordinados.

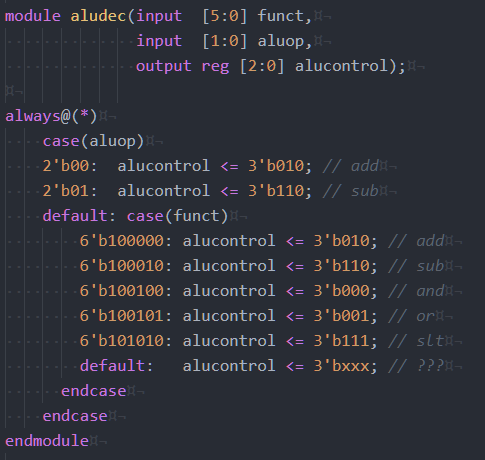
**1 - Main Decoder Output Table:**

****

(Nota: Control Word representa únicamente al output del main\_decoder, no al de controller)

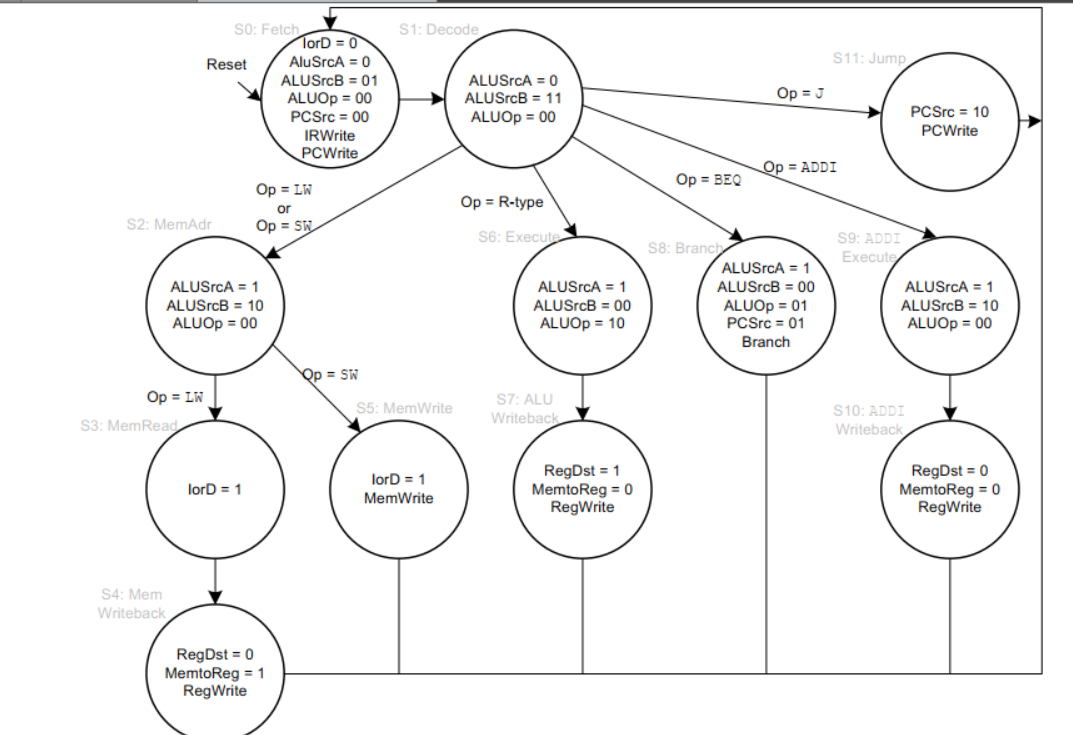
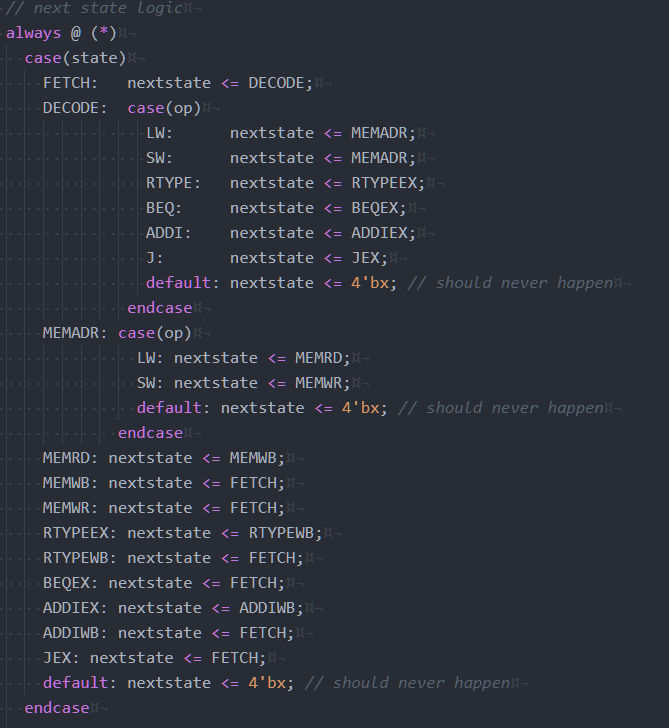
**2 - Módulos MCP:**

Módulo Aludec: Este módulo sigue siendo el mismo que en el laboratorio anterior (Single-cycle). No cuenta con las instrucciones ORI ni BNE.

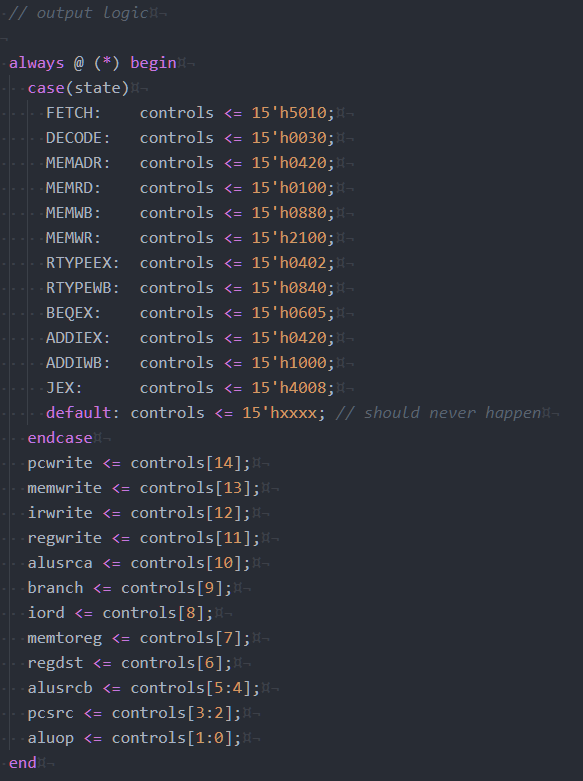


Módulo maindec: Este módulo es el que contiene la FSM del procesador. Dados los parámetros de cada fase, el state register y la decodificación de los dos primeros estados, S0 (Fetch) y S1 (Decode), se pide implementar los otros 9 estados así como el output logic de la máquina de Moore.

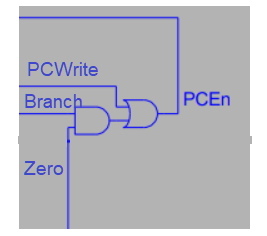
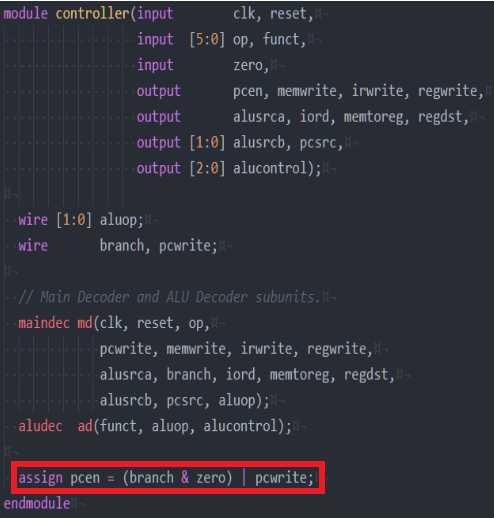
Nos limitaremos únicamente a mostrar partes del código que hemos tenido que diseñar por nuestra cuenta. En ese sentido, se tiene el next state logic que hemos implementado para las instrucciones faltantes:



Luego, se tiene el output logic, que vendrían a ser las 15 señales respectivas a los estados que identificamos en la tabla 1:



Módulo controller: Este módulo no es muy diferente a la versión final. Salvo los ajustes al pasarlo de SystemVerilog a Verilog, nuestra única modificación es implementar la lógica de pcen de acuerdo a la Figura 1

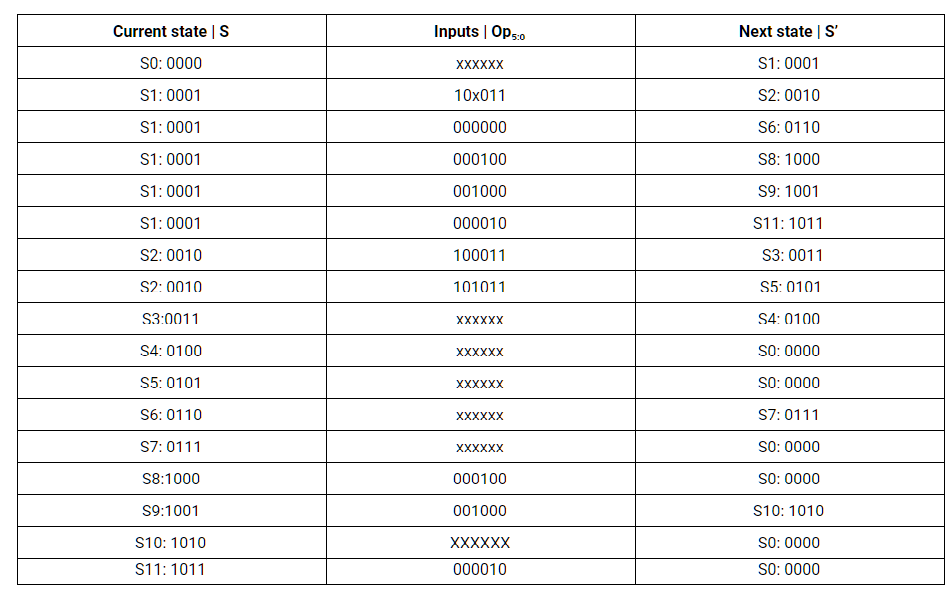


Aunque el código implementado es autoexplicativo, hemos optado por hacer explícito lo que hemos diseñado. Para ello, se han elaborado tablas que resumen lo solicitado. Ha terminado siendo muy útil contar con ellas en el momento de insertar el código.

*Table 2: State encoding of the Main decoder FSM*

|  |  |
| --- | --- |
| **State** | **Encoding S3:0** |
| S0: Fetch | 0000 |
| S1: Decode | 0001 |
| S2: MemAdr | 0010 |
| S3: MemRead | 0011 |
| S4: Mem Writeback | 0100 |
| S5: MemWrite | 0101 |
| S6: Execute | 0110 |
| S7: ALU Writeback | 0111 |
| S8: Branch | 1000 |
| S9: ADDI Execute | 1001 |
| S10: ADDI Writeback | 1010 |
| S11: Jump | 1011 |

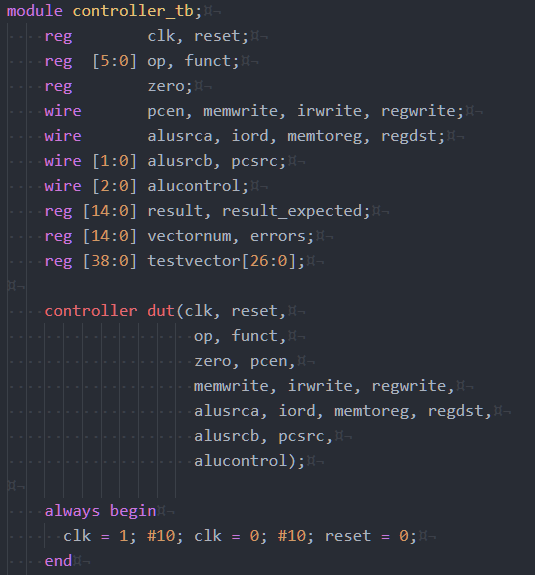
*Table 3: State transition with binary encoding*

****

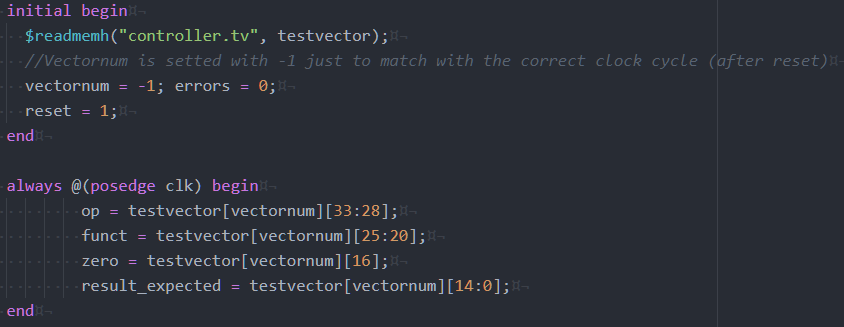
**3 - Controllertest\_xx testbench:**

En esta ocasión, explicaremos más detenidamente el testbench, puesto que es de las pocas veces durante el proyecto que lo hemos implementado prácticamente desde cero y usando testvectors.

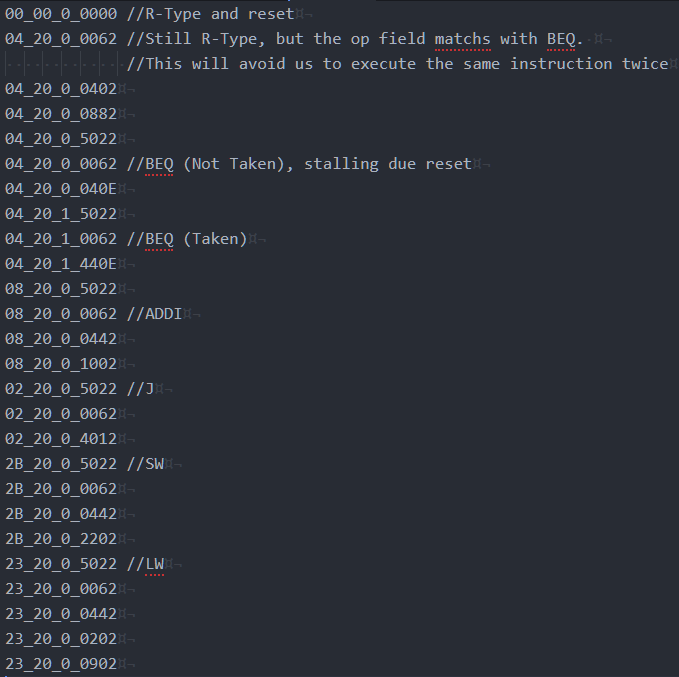
Lo primero viene a ser la inicialización del device under test, del clock y de nuestro reset.



Después, excitamos la matriz testvector usando instrucciones generadas por nosotros. Esto lo hacemos a través de controller.tv, que contiene un total de 7 instrucciones que deberán ser cumplidas luego de 25 ciclos de reloj (reset incluido). Adicionalmente, creamos una variable que contabilizará los errores de simulación. Finalmente, decodificamos la instrucción asignándoles a nuestras variables el valor que les corresponda.

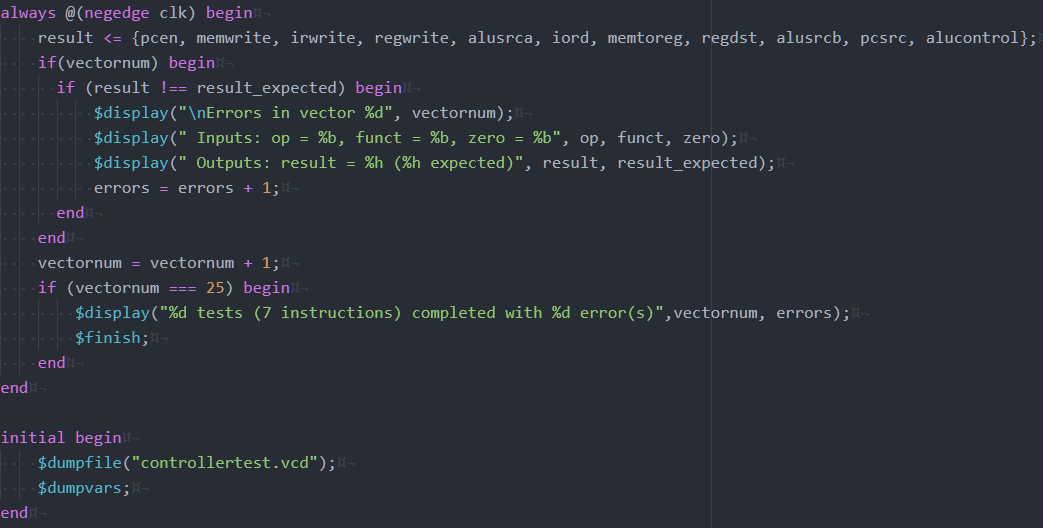


Las instrucciones que contiene nuestro .tv, con el cual probamos nuestra implementación, son:

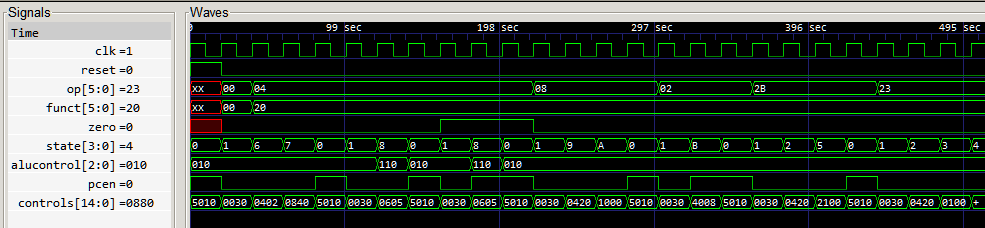


(Nota: Para evitar posibles errores de lectura, el .tv original no contiene ningún comentario. Insertamos el op de BEQ durante el reset para no perder esa instrucción en memoria)

Por último, comprobamos que el resultado sea el esperado. En caso de no serlo, nuestro contador errors irá aumentando. Imprimimos en la terminal la información necesaria para cada caso, generamos el .vcd respectivo y estamos listos para simular el programa.



**4 - Simulación del programa:**

****

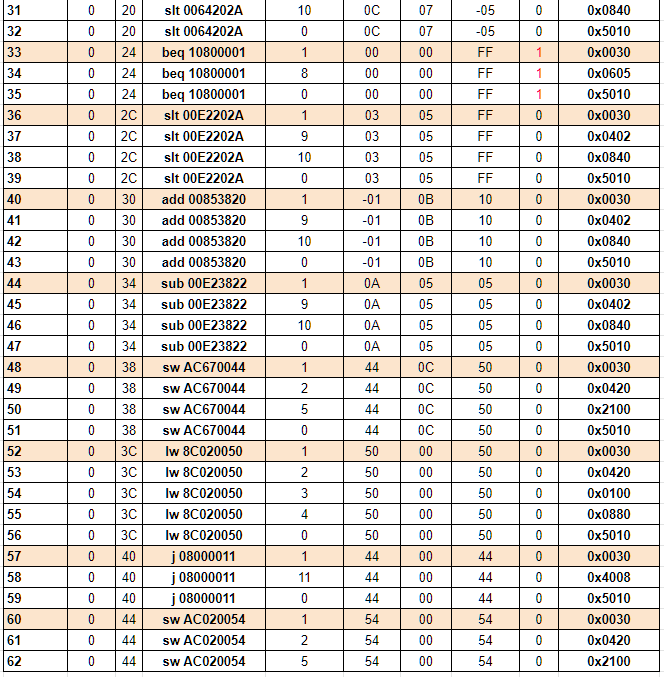
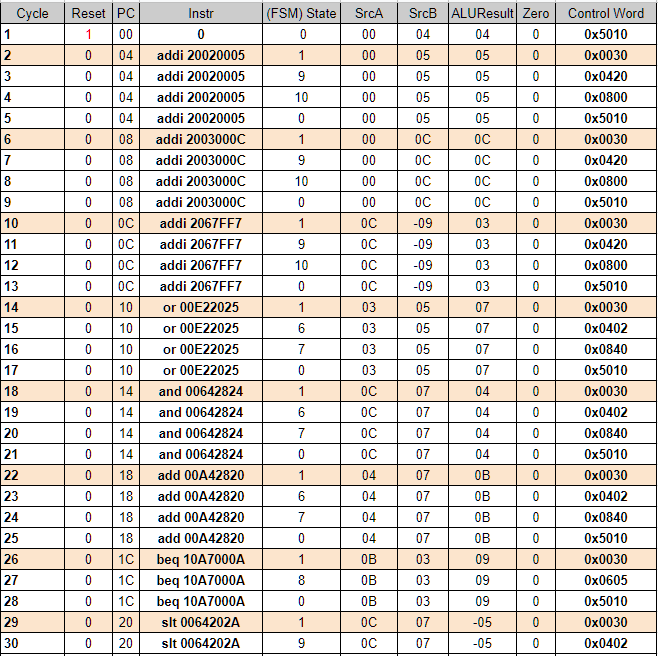
Podemos notar que la transición de estados es la esperada, así como la control word anexa a estos. La terminal también nos informa que no ha ocurrido ningún error en la simulación, lo que da cierre a la primera parte de la implementación del MIPS Multicycle.



**Lab 9: Multicycle Processor - Part 2: Datapath**

* Descripción de la tarea: Como última parte del proyecto final, se pide implementar las conexiones entre las distintas partes del microprocesador a través de un bloque structural, el llamado datapath.

**1 - Tabla 1 - Expected Outputs**

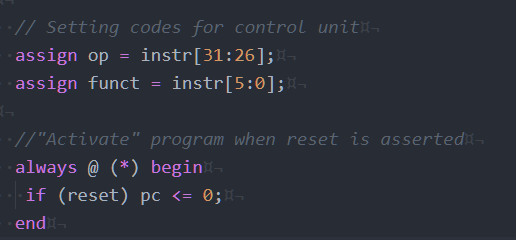


(Nota: Esta tabla y las demás se encuentran en sus respectivas carpetas según el laboratorio al que pertenecen)

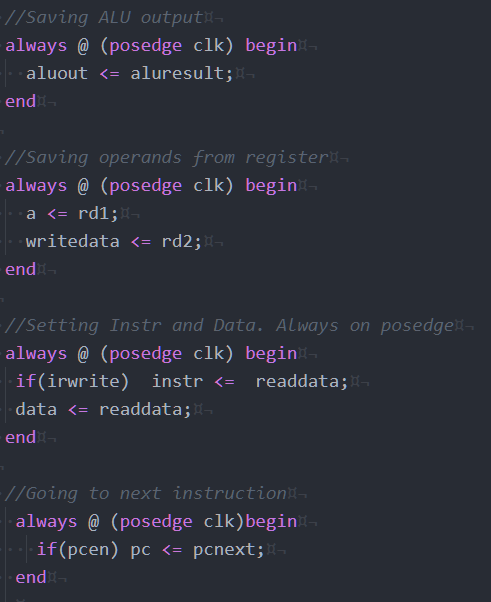
**2 - Structural Verilog Datapath:**

Se implementó el diagrama del MIPS Multicycle que se nos facilitó en el laboratorio 8. Este módulo contiene las conexiones entre las diferentes partes del procesador y se encarga de gestionar su flujo.

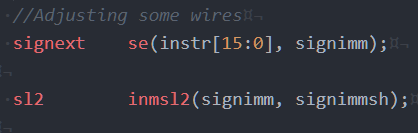
Lo primero es proporcionarle al control unit las señales que necesita decodificar para asignar las señales. Adicionalmente, generamos la condición de reset para inicializar el pc en 0.



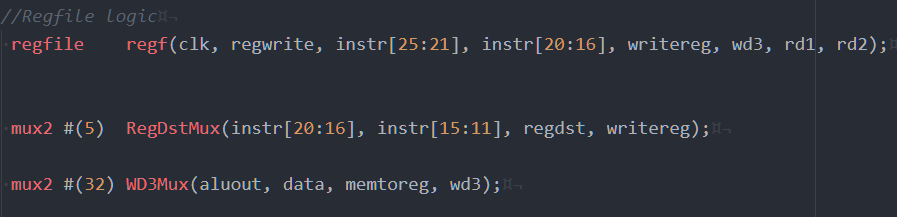
Luego, durante el posedge del clock signal, refrescamos y asignamos las respectivas para el siguiente ciclo



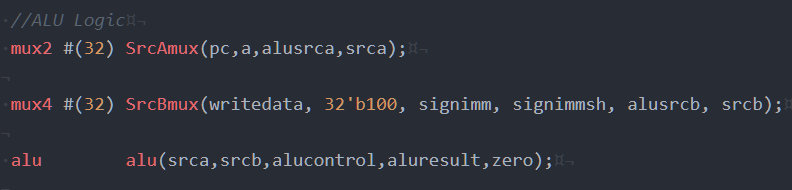
Debemos encargarnos de extender los 16 últimos bits de la instrucción para tenerlos hábiles si la instrucción así lo requiere. Algunas instrucciones requieren también que este pase por el módulo sl2.



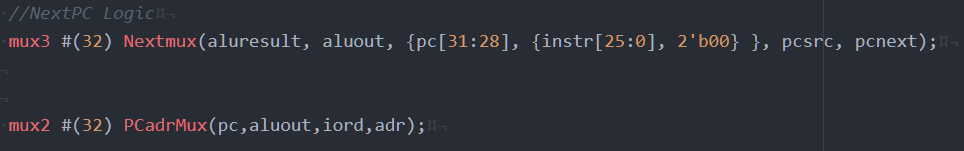
A continuación llamamos al regfile para gestionar nuestra data memory. Este módulo almacena valores y los sobrescribe si la instrucción así lo amerita. También tenemos un mux de 5 (log2 de 32, puesto a que representan posiciones en memoria) para hacer fetch de los operandos. Luego, tenemos un mux que se encarga de hacer writeback para algunas operaciones del ALU.



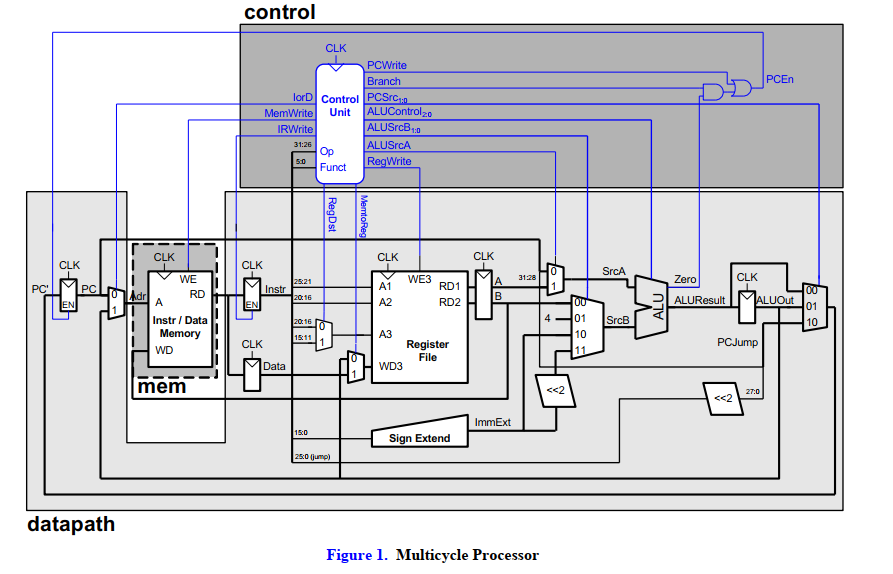
En relación a nuestro ALU, requerimos de dos multiplexores de 32 bits. El primero para filtrar si el primer operando será el operando que se solicita en la instrucción o el pc (en este último caso, se efectúa un pc + 4). El segundo mux hace lo propio con el segundo operando, que esta vez puede ser el operando b, un 4 decimal, para sumarlo al pc y continuar leyendo el .dat o cualquiera de las dos señales extendidas, ya sea la sign extend o la sign extend shifted by 2. Y, por último, claramente debemos llamar al ALU per se.



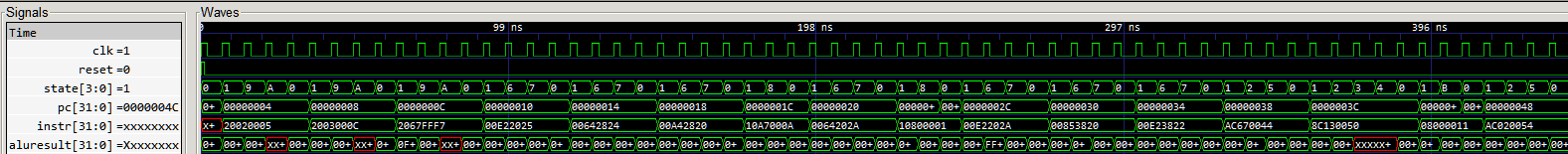
Finalmente, tenemos la lógica del próximo pc. El primer mux nos dirá si el pc next será igual a pc+4, pc+4+branch o un jump que se concatena en base a los primeros 4 bits del pc actual, los 25 bits de un immediate en J-instructions y 2 bits de 00 para que el valor sea word-aligned. También tenemos otro multiplexor que maneja la lógica del next address.



Recordar el diagrama que se usó para la implementación:

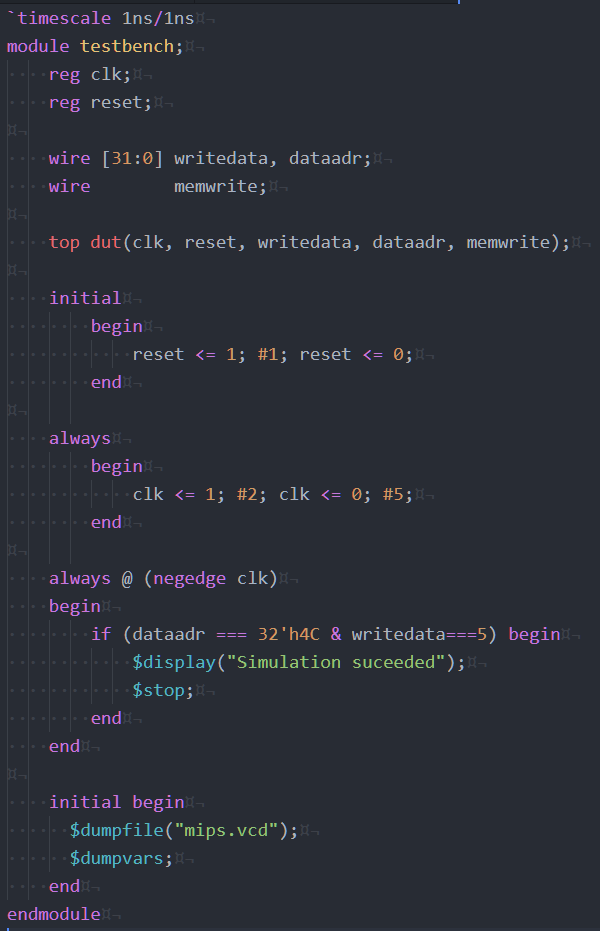


**3 - Simulación del programa:**

****

Como podemos observar, la lectura de instrucciones, lógica de pc, salida de aluresult y la transición de estados es la correcta.

Para esta simulación, se realizó el siguiente testbench:



Con esto, damos por concluido el laboratorio 9 y, consecuentemente, el proyecto solicitado.

**Bibliografía**

* Harris, D. & Harris, S. (2012). Digital Design and Computer Architecture. 2nd Edition. San Francisco, Morgan Kaufmann.