Universidad de Costa Rica

Escuela de Ciencias de la Computación e Informática

CI-0120 Arquitectura de computadoras

Examen 2

Profesor:

Francisco Arroyo Mora

Estudiante:

Josef Ruzicka González B87095

3 / 12 / 2021

A) (30 pts.) Pruebas de su CPU "pipeline" desarrollada durante el curso con las unidades de "forwarding" y "hazard detection".

Para poder llevar a cabo las pruebas propuestas en el enunciado de la prueba, se mejoró el "Forwarding Unit" que se había presentado anteriormente con la intención de que sirva para resolver "data hazards" por medio de la técnica de adelantamiento, la lógica que sigue esta unidad se representa en el siguiente pseudocódigo:

```
1a.  if(EX/MEM.RegWrite && EX/MEM.WriteReg != 0 &&
     EX/MEM.WriteReg == ID/EX.RegisterRs) ForwardA = 10


1b.  if(EX/MEM.RegWrite && EX/MEM.WriteReg != 0 &&
     EX/MEM.WriteReg == ID/EX.RegisterRt) ForwardB = 10
```

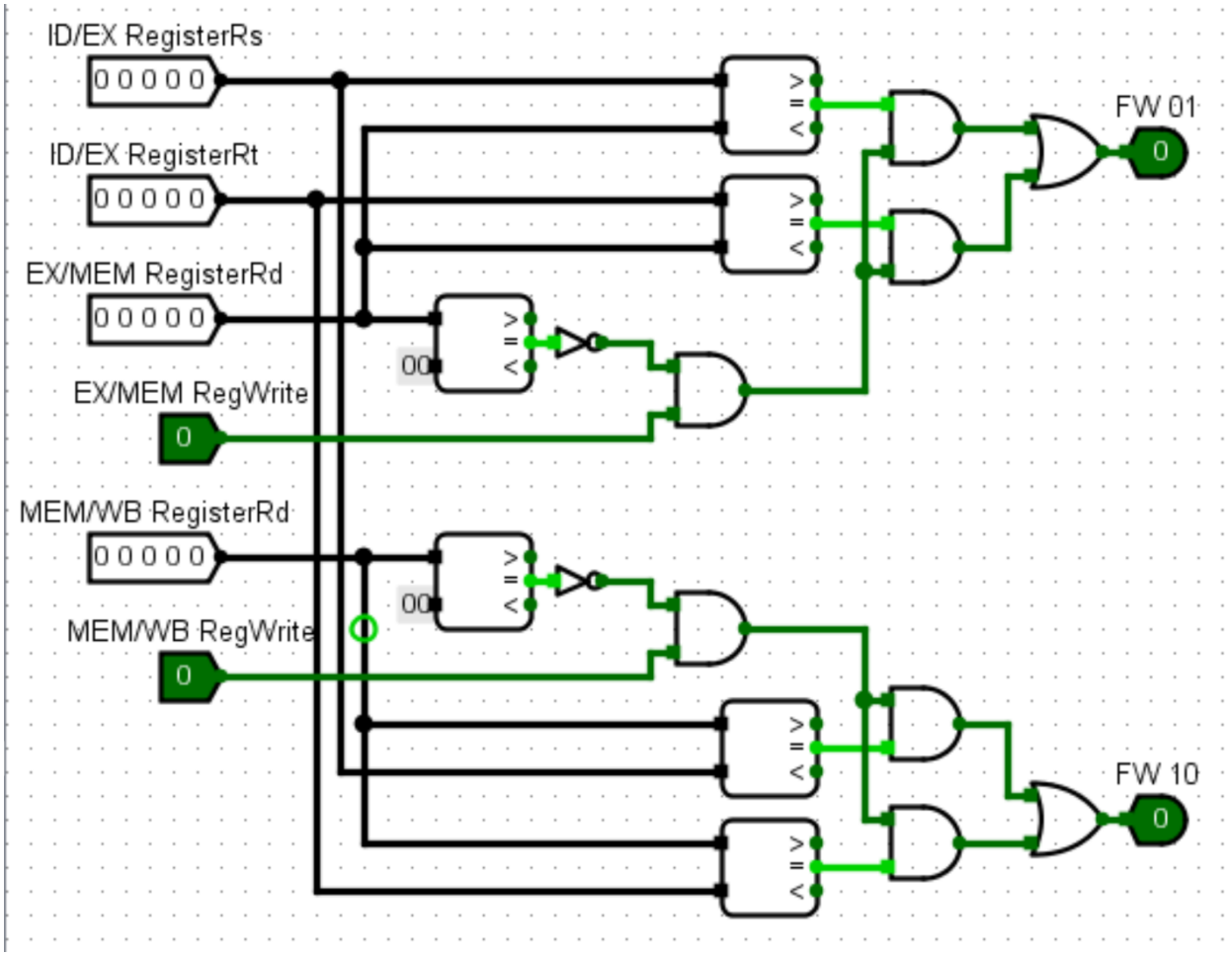
.

```
2a.  if(MEM/WB.RegWrite && MEM/WB.WriteReg != 0 &&
     MEM/WB.WriteReg == ID/EX.RegisterRs) ForwardA = 01


2b.  if(MEM/WB.RegWrite && MEM/WB.WriteReg != 0 &&
     MEM/WB.WriteReg == ID/EX.RegisterRt) ForwardB = 01
```

Pseudocódigos de soluciones a data hazards en forwarding unit tomadas de las filminas de "Lecture-9" de la sección de material/adicionales en el servidor de recursos del curso.

Con base al pseudocódigo presentado anteriormente, el diseño del circuito en Logisim se cableó como se puede observar en la siguiente imagen:



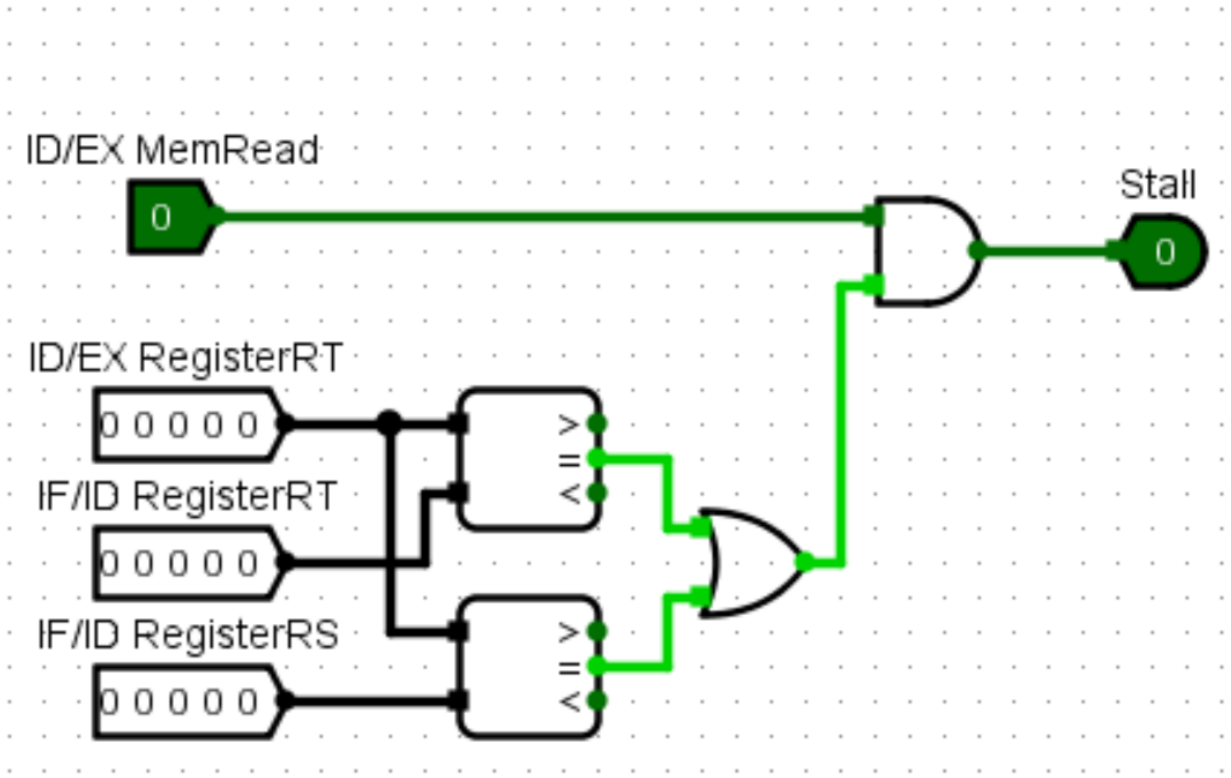
Circuito en Logisim del “Forwarding Unit”

Además de actualizar el “Forwarding Unit”, también se diseñó un circuito en logisim encargado de detectar los hazards para los cuales fuese necesario hacer una operación de “Stall” sobre el pipelining del circuito para de esta manera lograr dar el tiempo necesario para que los datos se encuentren en la forma que deben tener para que el programa del procesador se ejecute con éxito. Este circuito se llama un **“Hazard Detection Unit”** y su función corresponde al siguiente pseudocódigo:

```
if (ID/EX.MemRead &&
((ID/EX.RegisterRt == IF/ID.RegisterRs) ||
(ID/EX.RegisterRt == IF/ID.RegisterRt))) Stall the Pipeline
```

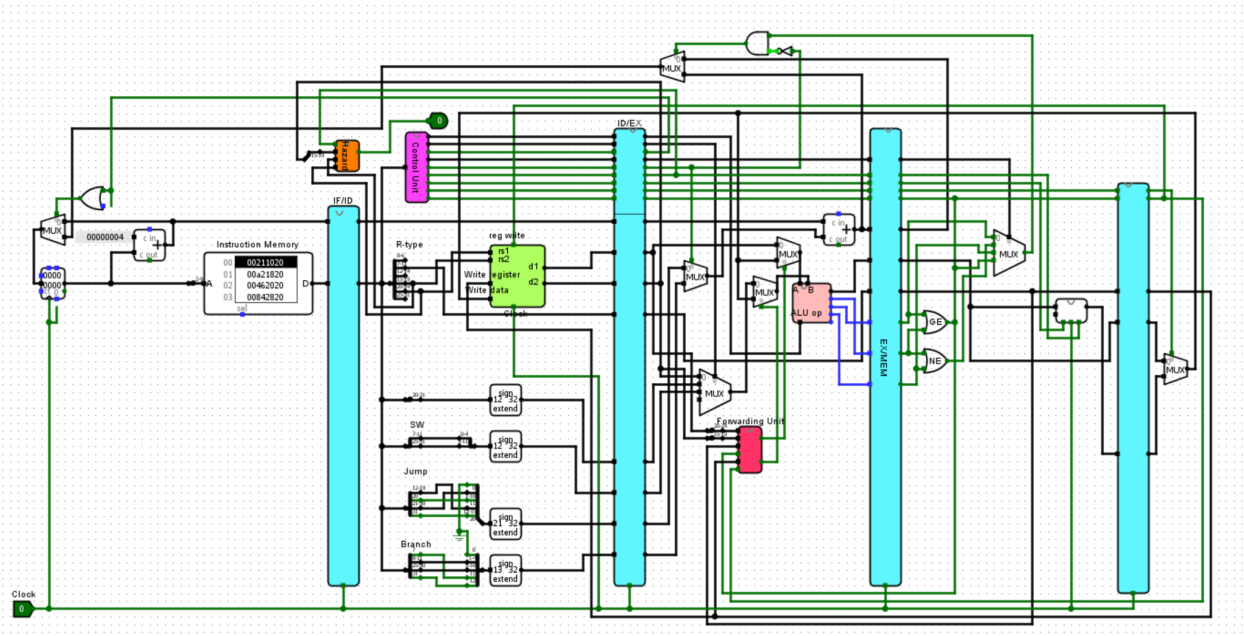
Pseudocódigo de la función del “Hazard Detection Unit”, tomado de las filminas de “Lecture-9” en la sección de material/adicionales del servidor de recursos del curso.

El circuito en Logisim de esta unidad se presenta de la siguiente manera:



Circuito en Logisim del “Hazard Detection Unit”

El Procesador RISC-V que integra los componentes “Forwarding Unit” y “Hazard Detection Unit” mostrados en las imágenes anteriores con el resto de componentes desarrollados a lo largo del curso y siguiendo la técnica de “Pipelining” presenta la siguiente forma:



Circuito en logisim procesador RISC-V con pipelining

Pruebas:

1) Ver “B87095_Prueba1.circ”
2) Ver “B87095_Prueba2.circ”

   Instrucciones:

   - lw $2, 22($1) =

     imm 000000010110, rs 00010, func3 010, rd 00001,op 0000011

     Hex: 01612083
   - add $4, $2, $3 =

     func7 0000000, rs 00010, rt 00011, func3 000, rd 00100, op 0110011

     Hex: 00218233
   - add $5, $2, $3 =

     func7 0000000, rs 00010, rt 00011, func3 000, rd 00101, op 0110011

     Hex: 002182b3
   - add $6, $4, $2 =

     func7 0000000, rs 00100, rt 00010, func3 000, rd 00110, op 0110011

Hex: 00410333

- slt $1, $4, $2 =
  func7 0000000, rs 00100, rt 00010, func3 010, rd 00110, op 0110011
  Hex: 00412333

3) Ver “B87095_Prueba3.circ”

Instrucciones:

- beq $1, $2, 8 =
  Imm 0 000000, rs 00001, rt 00010, func3 000, imm 1000 0, op 1100011
  Hex: 00110863
- and $4, $2, $3 =
  Func7 0000000, rs 00010, rt 00011, func3 111, rd 00100, op 0110011
  Hex: 0021F233
- or $5, $3, $4 =
  Func7 0000000, rs 00011, rt 00100, func3 110, rd 00101, op 0110011
  Hex: 003262B3
- add $6, $4, $5 =
  Func7 0000000, rs 00100, rt 00101, func3 000, rd 00110, op 0110011
  Hex: 00428333
- slt $7, $5, $6 =
  Func7 0000000, rs 00101, rt 00110, func3 010, rd 00111, op 0110011
  Hex: 005323B3
- lw $4, 22($3) =
  imm 000000010110, rs 00100, func3 010, rd 00011,op 0000011
  Hex: 01622183

Para ensamblar las instrucciones de las pruebas 2 y 3, se utilizó la tabla de códigos de operación de los procesadores RISC-V como ha sido utilizada a lo largo del curso, la misma puede ser leída a continuación.

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

RV32/RV64 *Zifencei* Standard Extension

imm[11:0]	rs1	001	rd	0001111	FENCE.I

RV32/RV64 *Zicsr* Standard Extension

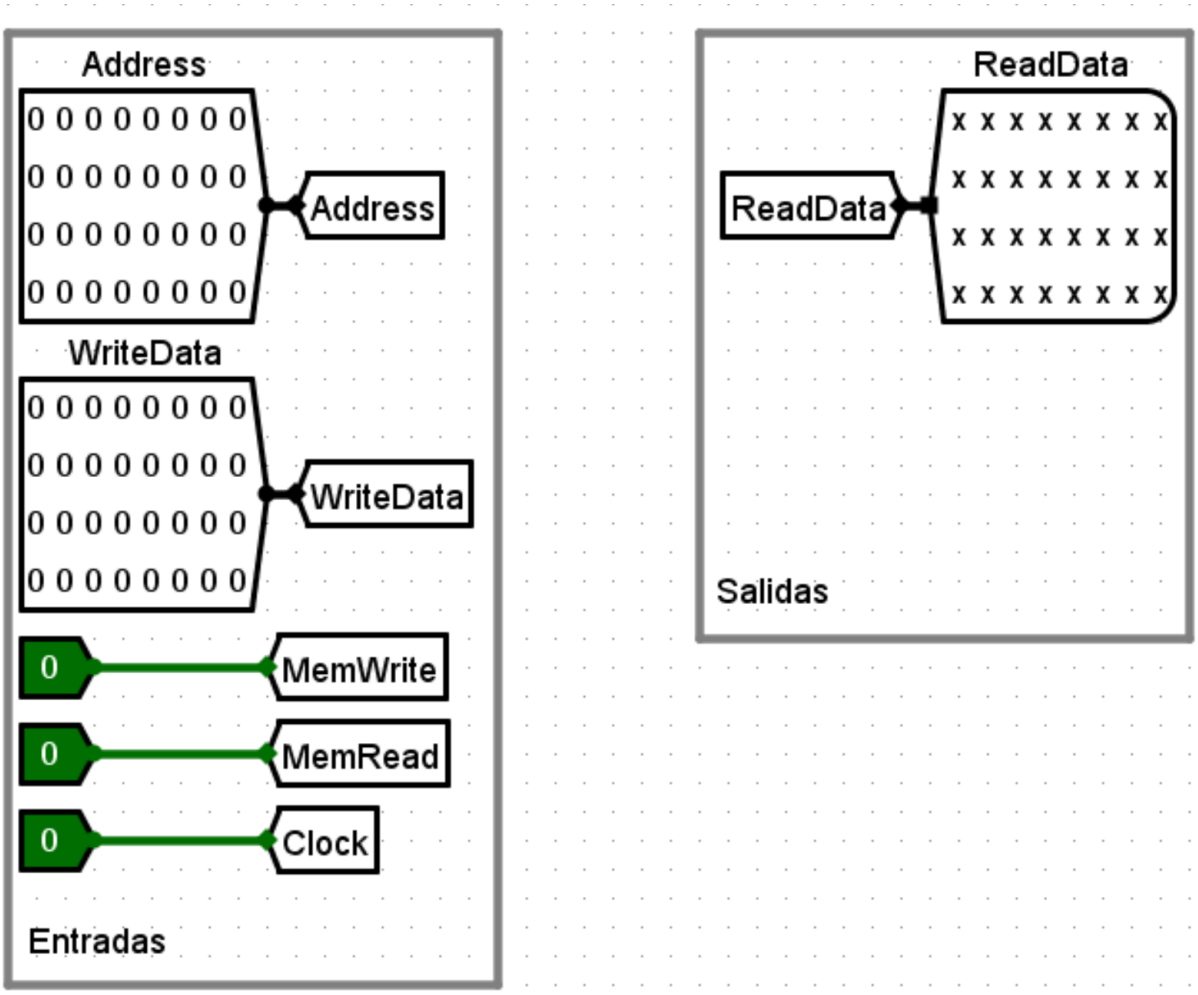
csr	rs1	001	rd	1110011	CSRRW
csr	rs1	010	rd	1110011	CSRRS
csr	rs1	011	rd	1110011	CSRRC
csr	uimm	101	rd	1110011	CSRRWI
csr	uimm	110	rd	1110011	CSRRSI
csr	uimm	111	rd	1110011	CSRRCI

B) (50 pts.) Construir en "logisim" una memoria para datos de 32 bits, pero que permita la lectura y escritura de bytes, medias palabras y palabras en su CPU.

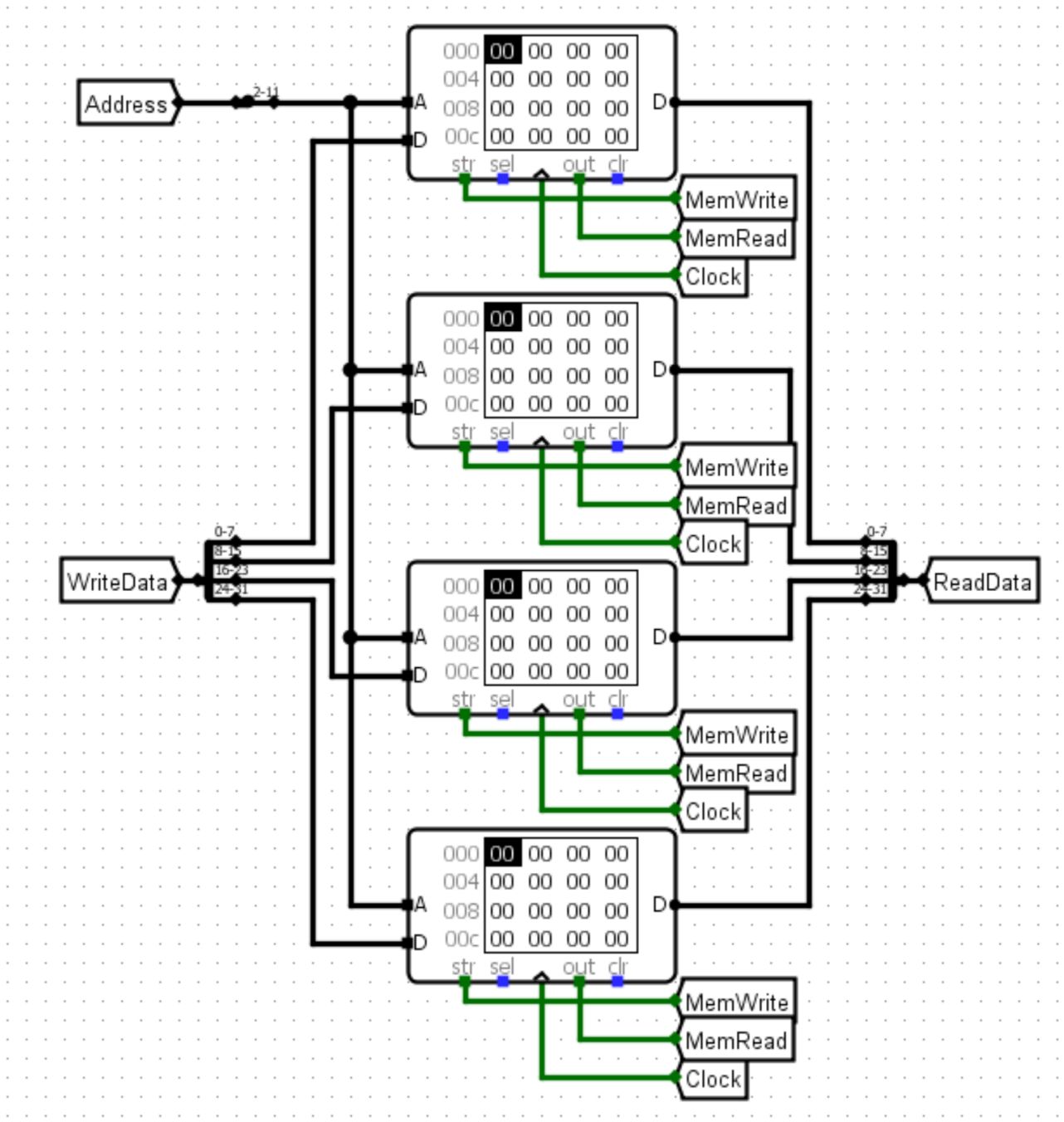
La "MemoryUnit" debe tener 2 entradas de 32 bits, una entrada **"Address"** que se referirá a la dirección de memoria que debe ser accesada, y una entrada **"WriteData"** que contendrá un valor de reemplazo cuando se desee sobrescribir el valor de una dirección de memoria.

Además de las entradas de 32 bits, se deben considerar tres entradas más de un único bit cada una: la entrada **"MemWrite"** que funciona como una bandera que se encarga de indicar si se debe escribir el valor de "WriteData" en la posición "Address" de la memoria, una entrada denominada **"MemRead"** que se utiliza para determinar si el valor presente en la dirección "Address" de la memoria debe ser almacenada en el pin de salida "ReadData" que será explicado en breve. La tercer entrada que se encarga de alimentar las unidades de memoria Ram con un bit será llamada **"Clock"** y se debe activar y desactivar (pasos que completan un ciclo de reloj) cada vez que en el circuito principal se envíe una señal de "relojazo" y entonces se llevará a cabo la actualización de memoria o la lectura que sea necesaria.

La salida **"ReadData"** mencionada anteriormente será una salida de 32 bits que será contenedora del valor presente en la dirección de memoria "Address" cuando así sea requerido e indicado por la bandera "MemRead". Las unidades de memoria serán cuatro memorias RAM de datos de 8 bits y con direcciones de 10 bits que toman sus respectivos datos de del "WriteData" por medio de un splitter que separa los 32 bits de los datos en grupos de 8 bits para cada una de las cuatro memorias, las direcciones se toman del "Address" con un splitter que tomará los bits del 2 al 11 que son los utilizados para determinar la dirección en la memoria. Para la salida de datos de las memorias también se utilizará otro splitter que tomará los 8 bits de la salida de cada unidad de memoria RAM y los juntará para formar el "ReadData" de 32 bits.



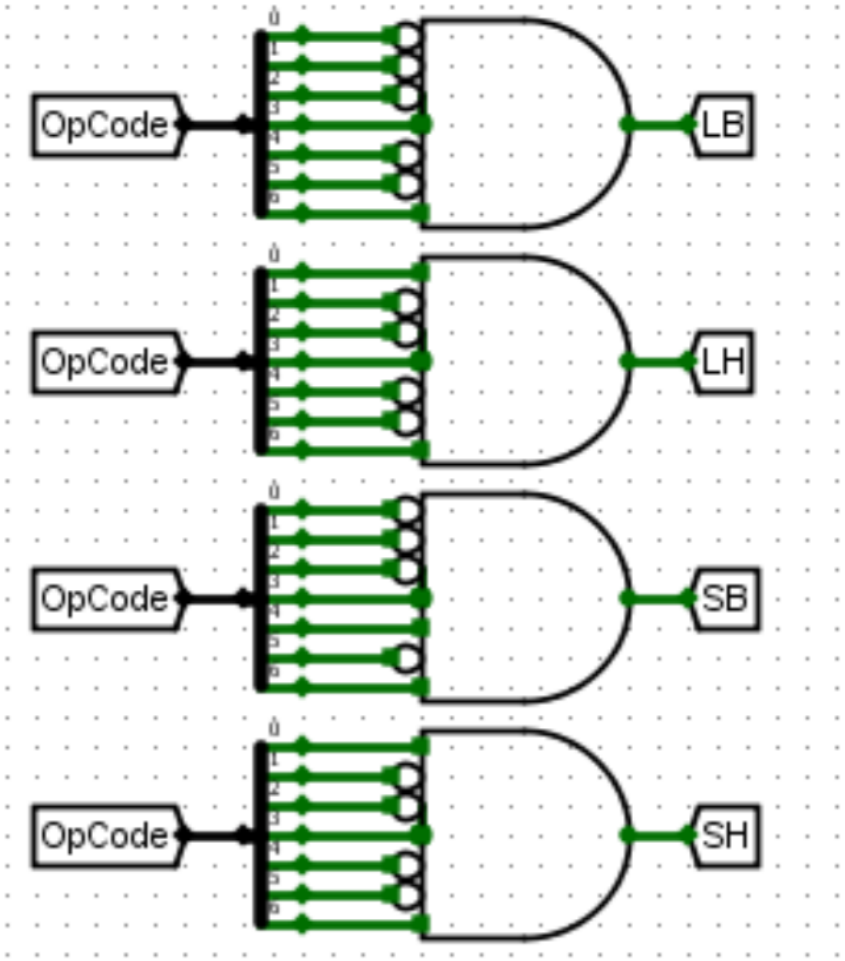
Entradas y salidas en logisim del “MemoryUnit”



Circuito en logisim del “MemoryUnit”

Para distinguir el tipo de dato que se desea utilizar en el “MemoryUnit” se modificó el “ControlUnit”, la modificación consistió en la adición de 4 salidas nuevas encargadas de indicar si el “OpCode” que entra a la unidad de control corresponde al código de operación de las instrucciones “SB” (Store byte), “SH” (Store Halfword), “LB”

(Load byte), “LH” (Load Halfword). Las instrucciones se diseñaron en el circuito de Logisim de la siguiente manera:



Modificación añadida al “ControlUnit” del CPU para que pueda ejecutar las instrucciones SB, SH, LB y LH.

C) (20 pts.) Completar los ejercicios solicitados:

3.1) [10] <3.1, 3.2> What is the baseline performance (in cycles, per loop iteration) of the code sequence in Figure 3.47 if no new instruction’s execution could be initiated until the previous instruction’s execution had completed? Ignore front-end fetch and decode. Assume for now that execution does not stall for lack

of the next instruction, but only one instruction/cycle can be issued. Assume the branch is taken, and that there is a one-cycle branch delay slot.

Latencies beyond single cycle	
Memory LD	+3
Memory SD	+1
Integer ADD, SUB	+0
Branches	+1
fadd.d	+2
fmul.d	+4
fdiv.d	+10

```
Loop:   fld    f2,0(Rx)
I0:     fmul.d f2,f0,f2
I1:     fdiv.d f8,f2,f0
I2:     fld    f4,0(Ry)
I3:     fadd.d f4,f0,f4
I4:     fadd.d f10,f8,f2
I5:     fsd    f4,0(Ry)
I6:     addi   Rx,Rx,8
I7:     addi   Ry,Ry,8
I8:     sub    x20,x4,Rx
I9:     bnz    x20,Loop
```

Figure 3.47 Code and latencies for Exercises 3.1 through 3.6.

instr		latency		branch delay slot
1	+	3	+	1
1	+	4		
1	+	10		
1	+	3		
1	+	2		
1	+	2		
1	+	1		
1	+	0		
1	+	0		
1	+	0		
1	+	1		

Baseline Performance in cycles, per loop = 38

Cada instrucción presente en el código ensamblado en la sección “Loop” requiere un ciclo para ser ejecutado, además se nos brinda para cada tipo de instrucción un valor de latencia que nos indica cuántos ciclos extra requiere cada una de las líneas de código en ser ejecutadas, para finalizar se nos aclara que debemos tomar en consideración un ciclo adicional ya que el Branch se encuentra ocupado.

Como es posible observar en la sección que representa una suma que se encuentra a la derecha de la “Figure 3.47”, se realizó la suma tal como es descrita en el párrafo anterior y el resultado de la suma de los ciclos tomados por las instrucciones y sus respectivas latencias fue 37 y a este se le agregó 1 ciclo más que corresponde al ciclo del “branch delay slot” mencionado al final del enunciado del problema.

5.1) [10/10/10/10/10/10/10] <5.2> For each part of this exercise, the initial cache and memory state are assumed to initially have the contents shown in Figure 5.37. Each part of this exercise specifies a sequence of one or more CPU operations of the form

`Ccore#: R, <address>` for reads

and

`Ccore#: W, <address> <-- <value written>` for writes.

For example,

`C3: R, AC10 & C0: W, AC18 <-- 0018`

Read and write operations are for 1 byte at a time. Show the resulting state (i.e., coherence state, tags, and data) of the caches and memory after the actions given below. Show only the cache lines that experience some state change; for example:

`C0.L0: (I, AC20, 0001)` indicates that line 0 in core 0 assumes an "invalid" coherence state (`I`), stores `AC20` from the memory, and has data contents `0001`. Furthermore, represent any changes to the memory state as `M: <address> <- value`.

Different parts (a) through (g) do not depend on one another: assume the actions in all parts are applied to the initial cache and memory states.

a. [10] <5.2> `C0: R, AC20`

b. [10] <5.2> `C0: W, AC20 <-- 80`

c. [10] <5.2> `C3: W, AC20 <-- 80`

d. [10] <5.2> `C1: R, AC10`

e. [10] <5.2> `C0: W, AC08 <-- 48`

f. [10] <5.2> `C0: W, AC30 <-- 78`

g. [10] <5.2> `C3: W, AC30 <-- 78`

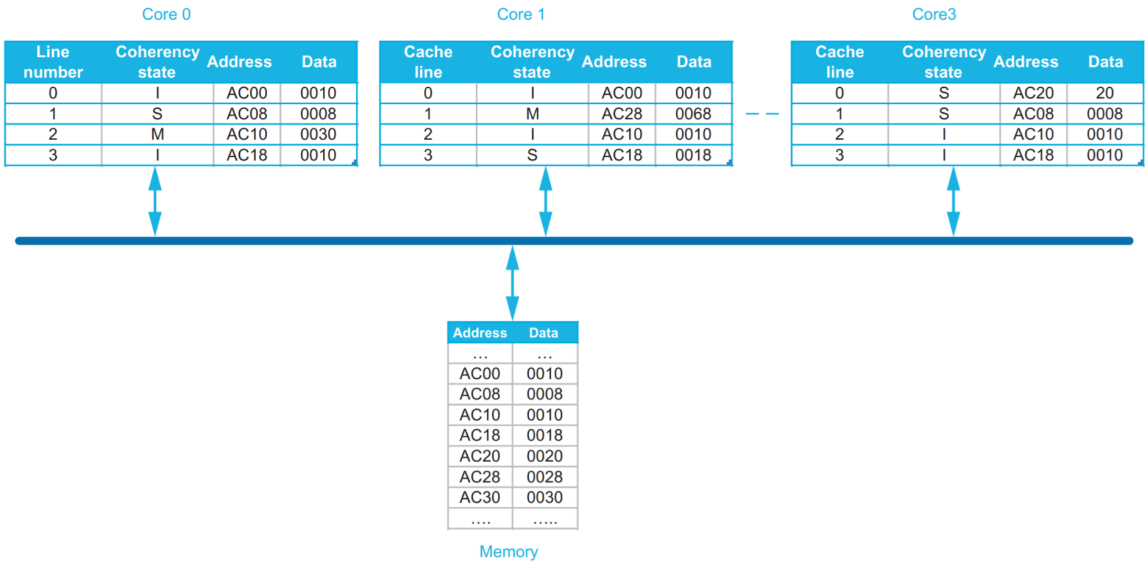


Figure 5.37 Multicore (point-to-point) multiprocessor.

Respuesta:

a) C0: R, AC20

C0.L0: (S, AC20, 0020)

b) C0: W, AC20 <-- 80

C0.L0: (M, AC20, 80)

C3.L0: (I, AC20, 20)

c) C3: W, AC20 <-- 80

C3.L0: (M, AC20, 80)

d) C1: R, AC10

C1.L0: (S, AC10, 0030)

e) C0: W, AC08 <-- 48

C0.L1: (M, AC08, 48)

C3.L1: (I, AC08, 0008)

f) C0: W, AC30 <-- 78

C0.L2: (M, AC30, 78)

g) C3: W, AC30 <-- 78

C3.L2: (M, AC30, 78)