Procesador con pipeline de cinco etapas

Jose Pablo Apú, B10407
Marco Torres, B16592
Chuan Wu, B27371

Estructuras de Computadores Digitales II

Escuela de Ingeniería Eléctrica
Universidad de Costa Rica

Resumen

En este documento se describe detalladamente el diseño, la implementación de un procesador con pipeline de cinco etapas.

I. Introducción

I. Desarrollo del proyecto

Se pretende describir un microprocesador con pipeline de cinco etapas. Las características principales son:

1. No tendrá direccionamiento indirecto.
2. Tendrá dos registros de uso general, llamados A y B.
3. No se harán operaciones de A con Memoria, ni de B con Memoria, únicamente entre A, B y constantes.
4. La memoria de datos estará separada de la de instrucciones.
5. Cada posición de la memoria de datos guardará un byte.
6. Cada posición de la memoria de instrucciones guardará dos bytes.
7. La memoria de datos tendrá 1024 posiciones (10 bits para indexarla).
8. La memoria de instrucciones tendrá 1024 posiciones (10 bits para indexarla).
9. No habrá subrutinas.
10. No se implementará atención a interrupciones.
11. No se implementará una pila.
12. Se supondrán únicamente números sin signo, por lo que no habrá bandera de rebase.

II. Plan de pruebas

Con la tabla de instrucciones, que se pueden observar en el cuadro 1, se piensa contruir pogramas pequeños, pero característicos, esto para observar el comportamiento del microprocesador.

II. Diseño

El que tenga un pipeline, permite estar ejecutando una intrucción, y que en ese momento pueda recibir otra intrucción. Sabemos que las intrucciones por lo general, necesitan de 2 a 3 ciclos de reloj. La idea es que al ejecutarse el segundo ciclo de reloj, el procesador pueda recibir la siguiente instrucción, sin que la primera instrucción finalice. Para lograr esto hay que definir muy bien cada bloque, así como tener mucho cuidado con las propagación de las señales.

I. Decodificador

El microprocesador tiene una unidad de control, que es este caso se llama decodificador. Este permite administrar las entradas que recibe la alu, los regirstros y además maneja los saltos del microprocesador.

Los principales componentes son:

II. ALU

Es el encargado se hacer operacioneces aritméticas y lógicas, además nos ayuda a pasar datos a la RAM. Es un circuito combinacional por lo que no ocupa de reloj.

III. Registros

Además de funcionar como un flip-flop. Este tiene como comportamiento predeterminado mantener su salida anterior. Pero esta puede ser cambiada por el decodificador.

IV. RAM y ROM

Ambas son memorias, nada más que la primera es solo lectura y la segundo si se puede escribir. Estan separadas, y la ROM es la que tiene el pograma ensamblado y la RAM es con el cual el microprocesador guarda y lee datos para hacer sus distintas operaciones.

III. Definición de la máquina

I. Multiplexores

Sistema de control

selM1	selM2	ln1	ln2
0	0	wA	imdt
0	1	wA	wB
1	0	imdt	imdt
1	1	imdt	wB

II. Acumulador

Sistema de control	
selX	wX
00	wX
01	inmdt
10	alu
11	mem

III. Decodificador

- **Entradas:**
  instr
- **Salidas:**
  selA, selB, selM1, selM2, inm, memDir, branchDir, jmpDir, jmpTaken, wrEnable, opCode
- **Asignaciónes:**
  inm = instr[0:7]
  memDir = jmpDir = instr[0:9]
  branchDir = instr[0:5]
  opCode = instr[10:15]

En la tabla 1 se observa las salidad de control, que hace el decodificador para manejar los registros y los muxes que van hacia la alu.

IV. Memoria de datos

- **Entradas:**

  wMemDirME, wALUME
- **Salidas:**

  wMemWB

Esta memoria presenta valores aleatorios al inicio del programa por lo que si se quiere declarar alguna posición de la memoria se deberá de hacer mediante el código a procesar, consta de un vector de registros que según sea la operación (escritura o lectura) guardará el valor de entrada en el registro o pondrá el dto de la posición desindexada en el cable de out.

V. Memoria de instrucciones

- **Entradas:**

  wPC

Salida según la intrucción

Codificación	Mnemónico	selA	selB	selM1	selM2	wrEnable
000 000	LDA	11	00	X	X	0
000 001	LDB	00	11	X	X	0
000 010	LDCA	01	00	X	X	0
000 011	LDCB	00	01	X	X	0
000 100	STA	00	00	0	X	1
000 101	STB	00	00	X	0	1
000 110	ADDA	10	00	0	1	0
000 111	ADDB	00	10	0	1	0
001 000	ADDCA	10	00	0	0	0
001 001	ADDCB	00	10	1	1	0
001 010	SUBA	10	00	0	1	0
001 011	SUBB	00	10	0	1	0
001 100	SUBCA	10	00	0	0	0
001 101	SUBCB	00	10	1	1	0
001 110	ANDA	10	00	0	1	0
001 111	ANDB	00	10	0	1	0
010 000	ANDCA	10	00	0	0	0
010 001	ANDCB	00	10	1	1	0
010 010	ORA	10	00	0	1	0
010 011	ORB	00	10	0	1	0
010 100	ORCA	10	00	0	0	0
010 101	ORCB	00	10	1	1	0
010 110	ASLA	10	00	X	X	0
010 111	ASRA	10	00	X	X	0
011 000	JMP	00	00	X	X	0
011 001	BAEQ	00	00	X	X	0
011 010	BANE	00	00	X	X	0
011 011	BACS	00	00	X	X	0
011 100	BACC	00	00	X	X	0
011 101	BAMI	00	00	X	X	0
011 110	BAPL	00	00	X	X	0
011 111	BBEQ	00	00	X	X	0
100 000	BBNE	00	00	X	X	0
100 001	BBCS	00	00	X	X	0
100 010	BBCC	00	00	X	X	0
100 011	BBMI	00	00	X	X	0
100 100	BBPL	00	00	X	X	0
100 101	NOP	00	00	X	X	0

Cuadro 1: Salidas de control, del decodificador

- **Salidas:**

  instID

La memoria está compuesta con una vector de registros que es instanciada con la función readmemb que toma los datos de un archivo externo y los introduce en caracter binario al vector de registro,dependiendo de la entrada wPC se tendrá en la salida la posición en el vector correspondiente a la línea de del siguiente código a procesar.

IV. Implementación

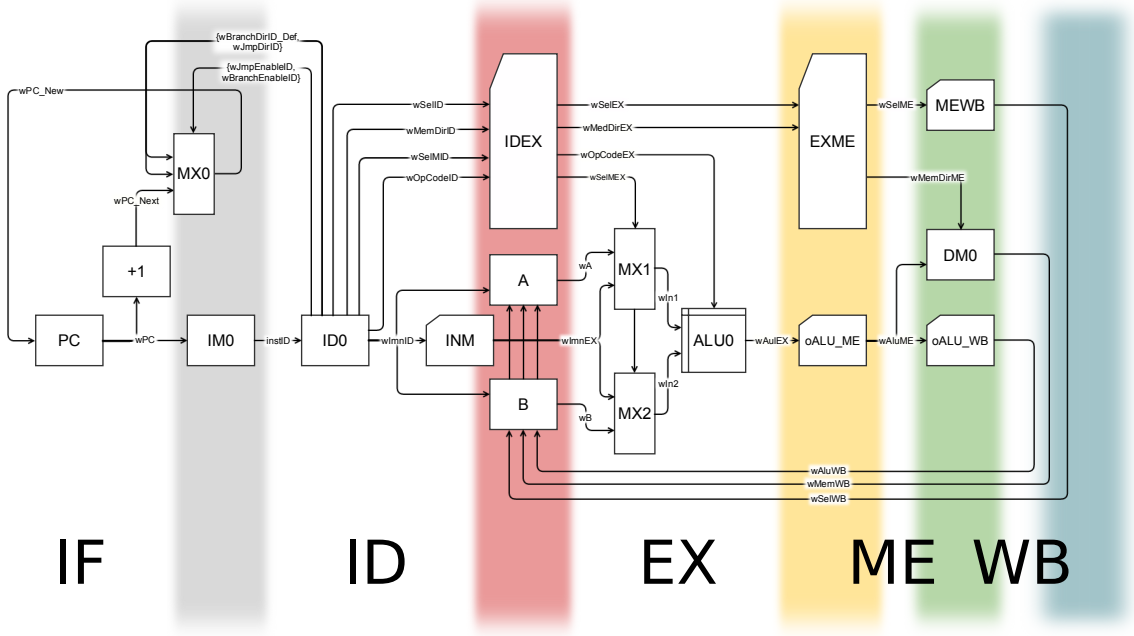

Figura 1: Plan de pruebas

V. Verificación y plan de pruebas

Asi como se menciona en la introducción para validar el diseño se ejecuto el plan de pruebas. Acontinuacion se listan cada una de las prubas que se realizaron una vez que se comprobó que cada elemento fuincionaba individualmente.

I. Operaciones aritméticas y de carga de constantes

Para esta prueba se propuso la creacion de un programa que tomara dos numeros constante, los restara y luego sumara otro. Acontinuacion se muestra el código en lenguaje ensamblador.

```
                #Programa 1:
NOP
LDCA 9A         #Se carga un $9A al acomulador A
NOP
NOP
LDCB 2C         #Se carga un $2C al acomulador B
NOP
NOP
SUBA            #(A) <- (A)-(B)
NOP
NOP
ADDCA 14        #(A) <- (A) + ($14)
```

NOP
NOP
###

En la figura 2 se muestra el resultado de correr este programa el tiempo suficiente para que todas las instrucciones pasen por las etapas.

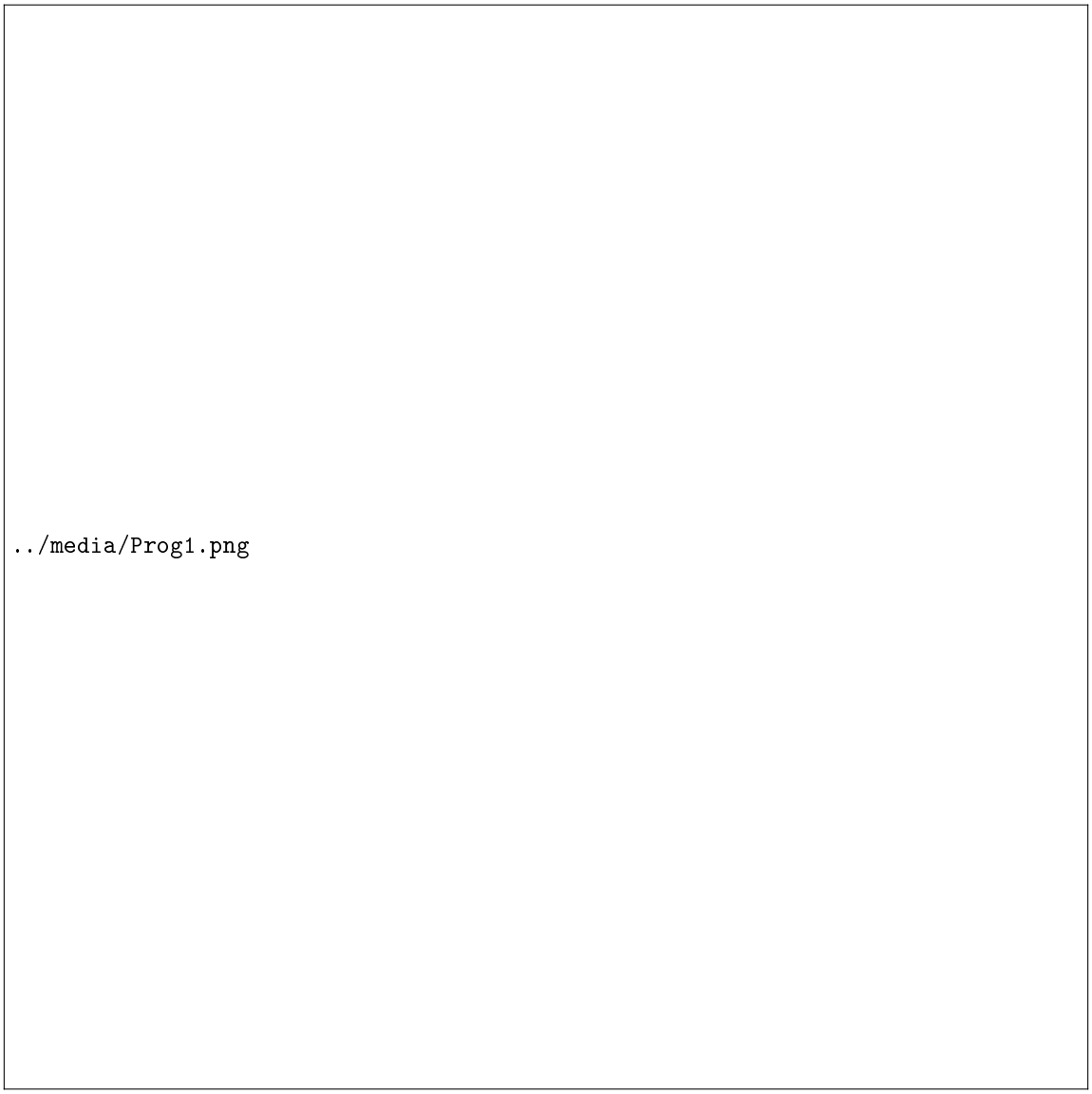

Figura 2: Señales de control y datos durante la ejecucion de programa 1.

Vale la pena prestar atencion a la linea **wOpCodeID** pues esta dice cual instrucccion se acaba de decodificar, tambien es importante ver a linea **wAluEX** pues esta muestra la salida de la *ALU*, en este caso los valores inmediatos que se van almacenar, finalmente al detallar las lineas **wA** y **wB** se puede ver el resultado de las instrucciones que se estan ejecutando. Adicionalmente aunque no

se muestren las demas señales de control se verifico que fueran las correctas, por lo cual podemos comprobar el resultado de la operacion 1.

$$(\$9A - \$2C) + \$12 = \$14 \tag{1}$$

II. Acceso a memoria (lectura y escritura)

Para probar el funcionamiento de las instrucciones de acceso a memoria basto con guardar el contenido del acomulador en una posicion de memoria y luego cargarlo desde la memoria a otro acomulador. Acontinuacion se muestra el código en lenguaje ensamblador.

```
                #Programa 2:
NOP
LDCA 9A                 #Se carga un $9A al acomulador A
NOP
NOP
STA 100                 #Se guarda el contenido de A
NOP                     #en la posicion $100 de la memoria
NOP
LDB 100                 #Se carga al acomulador B el
NOP                     #contenido de la posicion $100
NOP
                ###
```

En la figura 3 se muestran las señales al ejecutar este programa.

En este caso tambien es importante detallar las lineas **wA**, **wB wAluEX** y **wOpCode**, pero por el objetivo de la prueba se centro el analisis en las lineas **wWrEnableME** y **wMemDirME**. Para todos los casos la memoria siempre esta cargando en la salida el contenido de la posicion de memoria que indica wMemDirME pero solo escribe lo que esta su entrada cuando la linea wWrEnable esta en alto. Finalmente se puede comprobar que las instrucciones se ejecutaron bien observando la lina **wB** al final de la ejecucion.

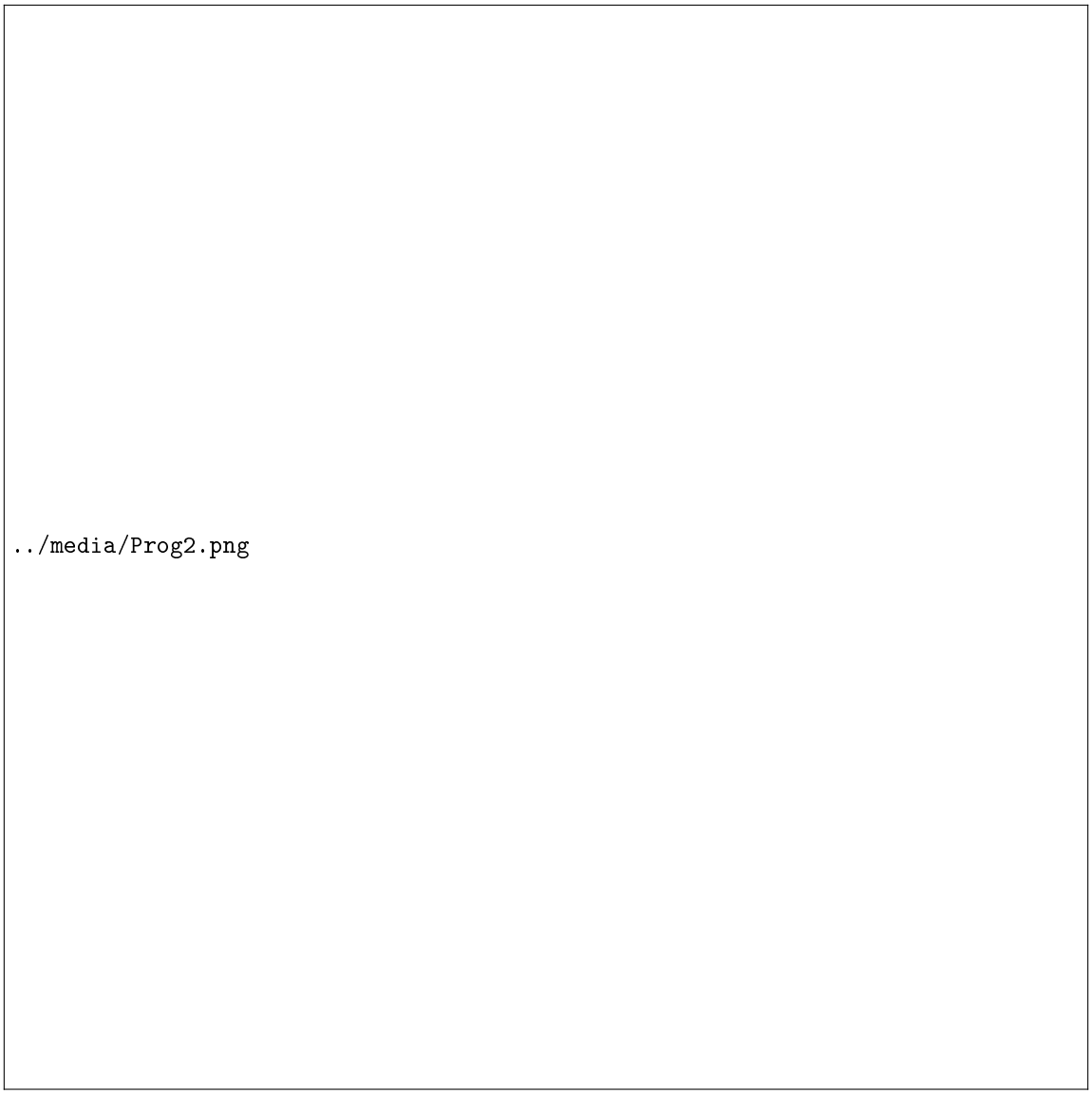

Figura 3: Señales de control y datos durante la ejecucion de programa 2.

III. Operaciones lógicas

En el caso de las operaciones lógicas se hizo un programa que tomara un numero y le quitara los cuatro bits menos significativos y luego desplazara los cuatro bits mas significativos cuatro veces a la derecha.Acontinuacion se muestra el código en lenguaje ensamblador.

```
                    #Programa 3:
NOP
LDCA 9A         #Se carga un $9A al acomulador A
NOP
NOP
LDCB F0         #Se carga un $F0 al acomulador B
NOP
NOP
ANDA            #Se hace un AND bit por bit entre los acomuladores
NOP
NOP
ASRA            #Se rota una vez a la derecha el acomulador A
NOP
NOP
ASRA
NOP
NOP
ASRA            #Se rota una vez a la derecha el acomulador A
NOP
NOP
ASRA            #Se rota una vez a la derecha el acomulador A
NOP
NOP
                    ###
```

En la figura 4 se muestran las señales al ejecutar este programa.

Aqui solamente vale la pena resaltar el contenido en las lineas **wOpCodeID** que muestra la instruccion a ejecutar y la linea **wA** que muestra el resultado del contenido del acomulador despues de ejecutar las intrucciones lógicas. Ademas es posible confirmar el resultado haciendo los calculos a mano 2.

$$\$9A(AND)\$F0 = \$90 \tag{2}$$

$$(\$90) >> 4 = \$09 \tag{3}$$

IV. Control de flujo | Saltos incondicionales

Para los saltos incondicionales solo se tuvo que interrumpir la linea del contador de programa y que siguiera en otro lado, por eso se hizo un programa que cargara un numero al acomulador A y mas adelante lo alterara pero antes de eso se coloco un salto incondicional de modo que si el salto funcionaba el dato no seria alterado. Acontinuacion se muestra el código en lenguaje ensamblador.

```
#Programa 4:
```

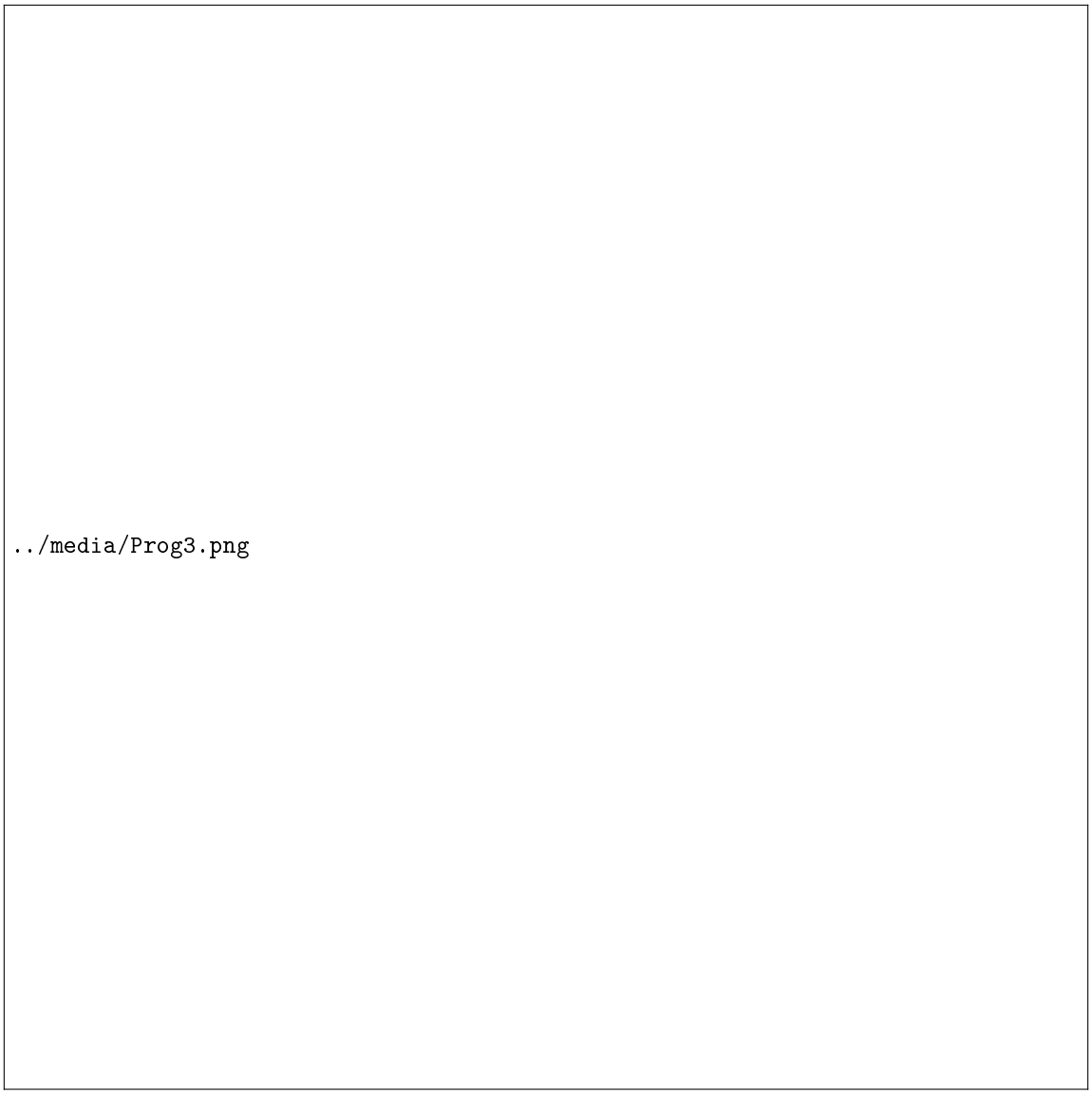

Figura 4: Señales de control y datos durante la ejecucion de programa 3.

```
NOP
LDCA 05                 #Se carga al acomulador A un $05
NOP
NOP
JMP J1 :                #Se salta a la posicion indicada por J1
NOP
NOP
LDCA FF                 #Se carga al acomulador A un $FF
NOP                     #(este codigo no se ejecuta)
NOP
: J1 :                  #Etiqueta de Salto 1
NOP
NOP
NOP
NOP
NOP
NOP
                ###
```

En la figura 5 se muestran las señales al ejecutar este programa.

Para esta prueba es de mucha importancia observar las lineas **wPC**, **wJmpEnableID** y **wJmpDirID** porque si bien la linea de direccion de salto siempr mustra algun valo solo se salta a este cuando el habilitador esta arriba. Se puede ver facilmente que la instruccion funciona bien al observar el contador de programa en **wPC**.

V. Control de flujo | Saltos condicionales

Para los saltos condicionales se hizo un programa que cargara un numero en el acomulador A y otro en el acomulador B. Luego desplazaria hacia la izquierda el numero en A tantas veces como induque el numero en B. Acontinuacion se muestra el código en lenguaje ensamblador.

```
                #Programa 5:
NOP
LDCA 01                 #Se carga un 1 en el acomulador A
NOP
NOP
LDCB 3                  #Se carga un 3 en el acomulador B
: J2 :                  #Etiquita de Salto 2
NOP
NOP
ASLA                    #Se desplaza hacia la izquierda el numero en A
NOP
NOP
SUBCB 1                 #Se decrementa el numero en B
NOP
NOP
NOP
BBNE J2 :               #Si el numero en B no es cero se
NOP                     #salta a la etiqueta
```

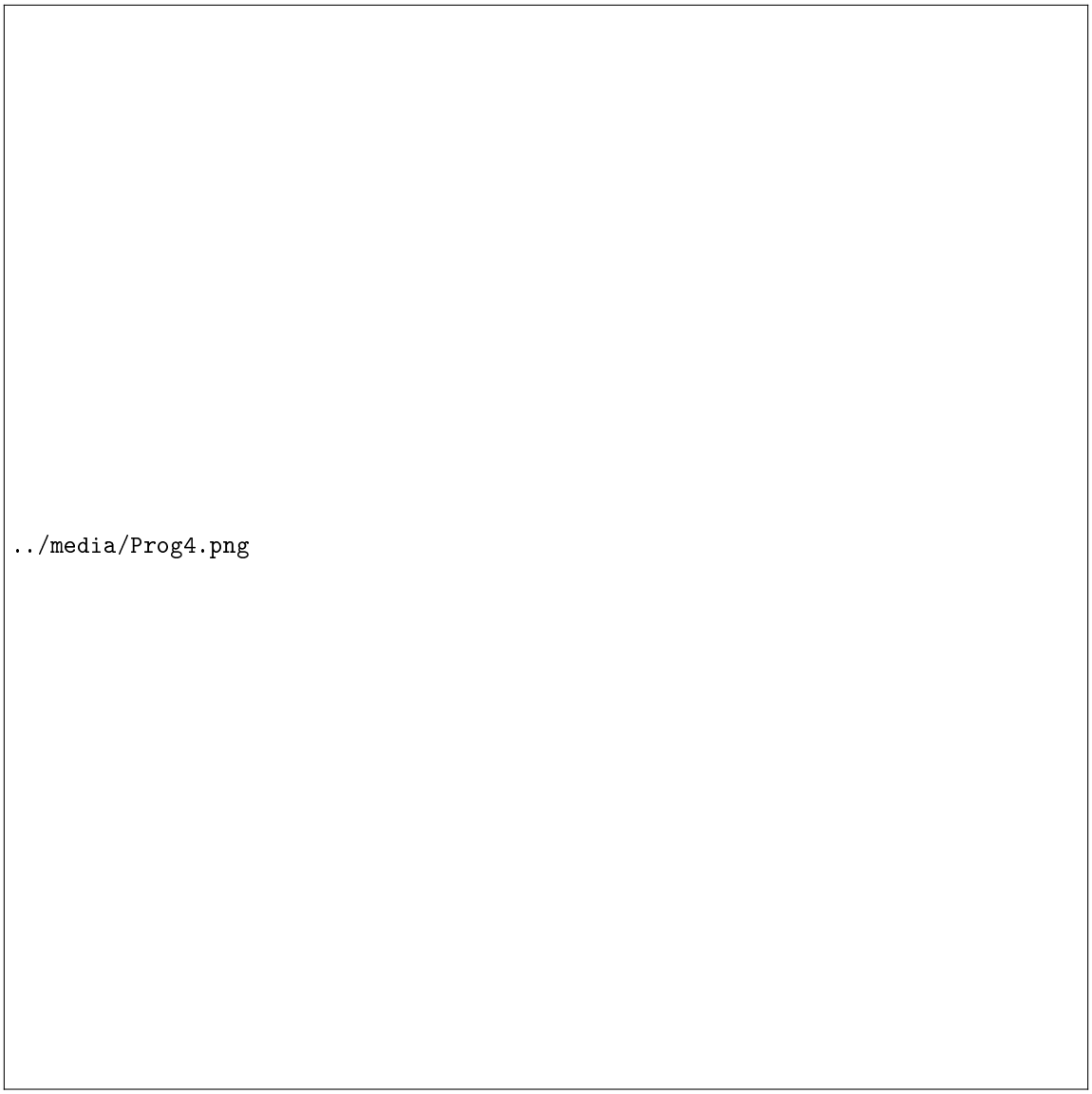

Figura 5: Señales de control y datos durante la ejecucion de programa 4.

NO

###

En la figura 6 se muestran las señales al ejecutar este programa.

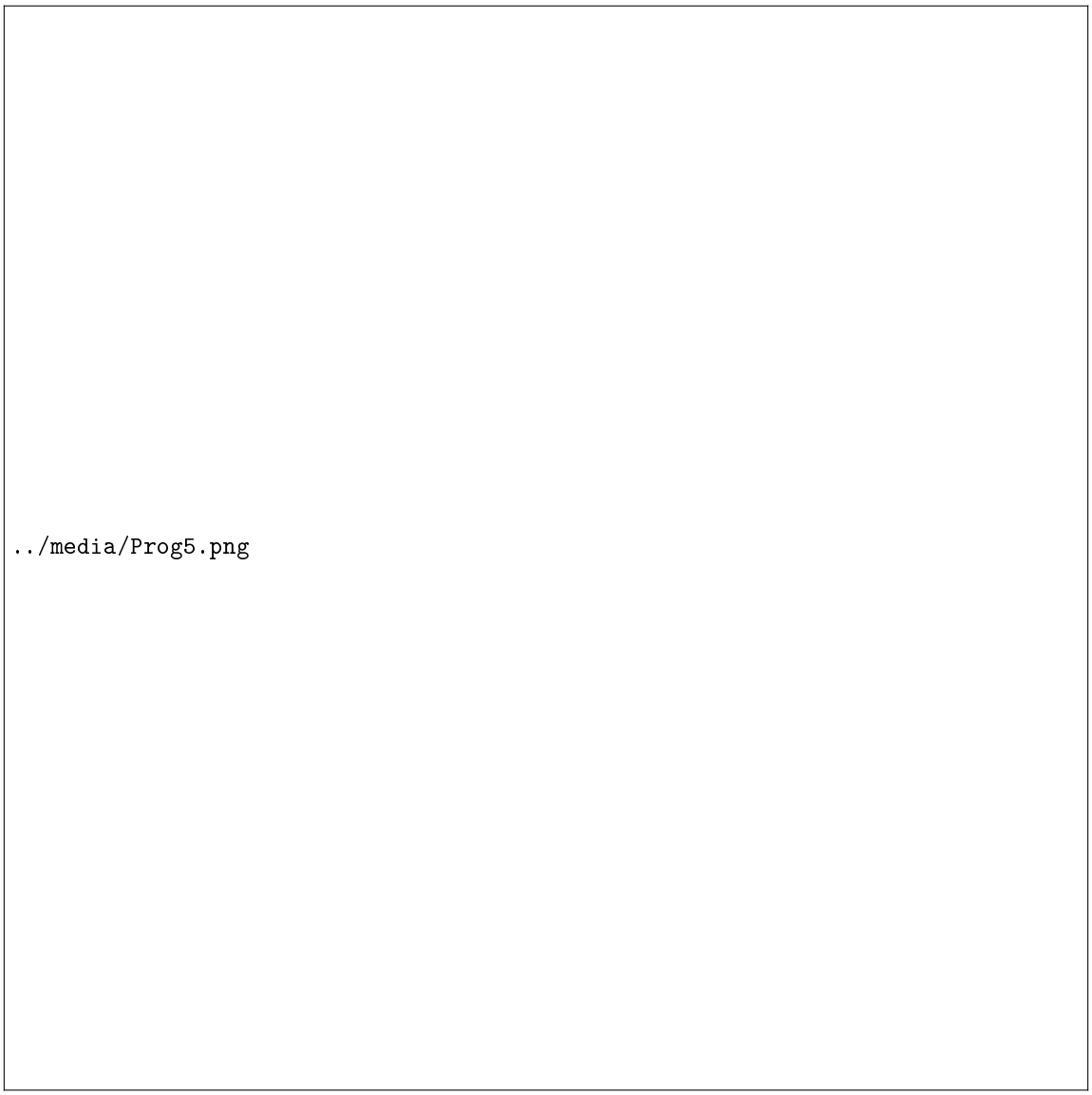

Figura 6: Señales de control y datos durante la ejecucion de programa 5.

Para verificar el funcionamiento de este programa se deben observar las lineas **wPC**, **wA**, **wB** y **wBranchEnableID** y facilmente se puede observar el resultado que se muestra en la linea wA para verificar el resultado 4.

$$(\$01) >> 3 = \$08 \tag{4}$$

EN otra prueba.. CHUAN AQUÍ VA UNA SUYA

VI. Conclusiones y recomendaciones

- Apezar de que no se exploto al máximo el "pipeline"se logro diseñar e implementar un procesador completamente funcional que ejecutara programas y almacenara los resultados.

- Como en el diseño solo se implemento con las señales de control y datos basicas el procesador no puede evitar o reducir "hazards"de ningun tipo, esto por falta de tiempo para implementar esta funcionalidad. A pesar de esto aqui se comenta como se planeo ampliar el diseo para soportar esta funcionalidad.

- Al ejecutar una instrucccion de salto, ya sea condicional o incondicional el contador de programa se incrementa pues aun no se ha decodificado la instruccion, esto causa que se lea la siguiente instruccion despues de la de salto. Para solucionar esto se planeo implementar un modulo que controlara el contenido de los registros de "pipe", de este modo cuando se leyera la siguiente instruccion el modulo reiniciaria el registro de "pipeÏF/ID para convertir la instrucccion leida en un NOP.

- Tambien por la forma en la que se diseño la etapa WB, si se ejecuta una instruccion que lea el acomulador A luego de una que lo escriba el valor leido sera un valor viejo. Para preveer este problema se debe crear un modulo que controle los registros de "pipe"de modo que pueda detener algunas etapas para empezar a introducir burbujas en las etapas que tienen el problema.

- Finalmente vale la pena agregar que aunque puede llegar a ser muy complejo diseñar un procesador, en general si se modulariza de la forma correcta y se verifica cada parte por separado la mayor parte del diseño se vuelve facil de diseñar.