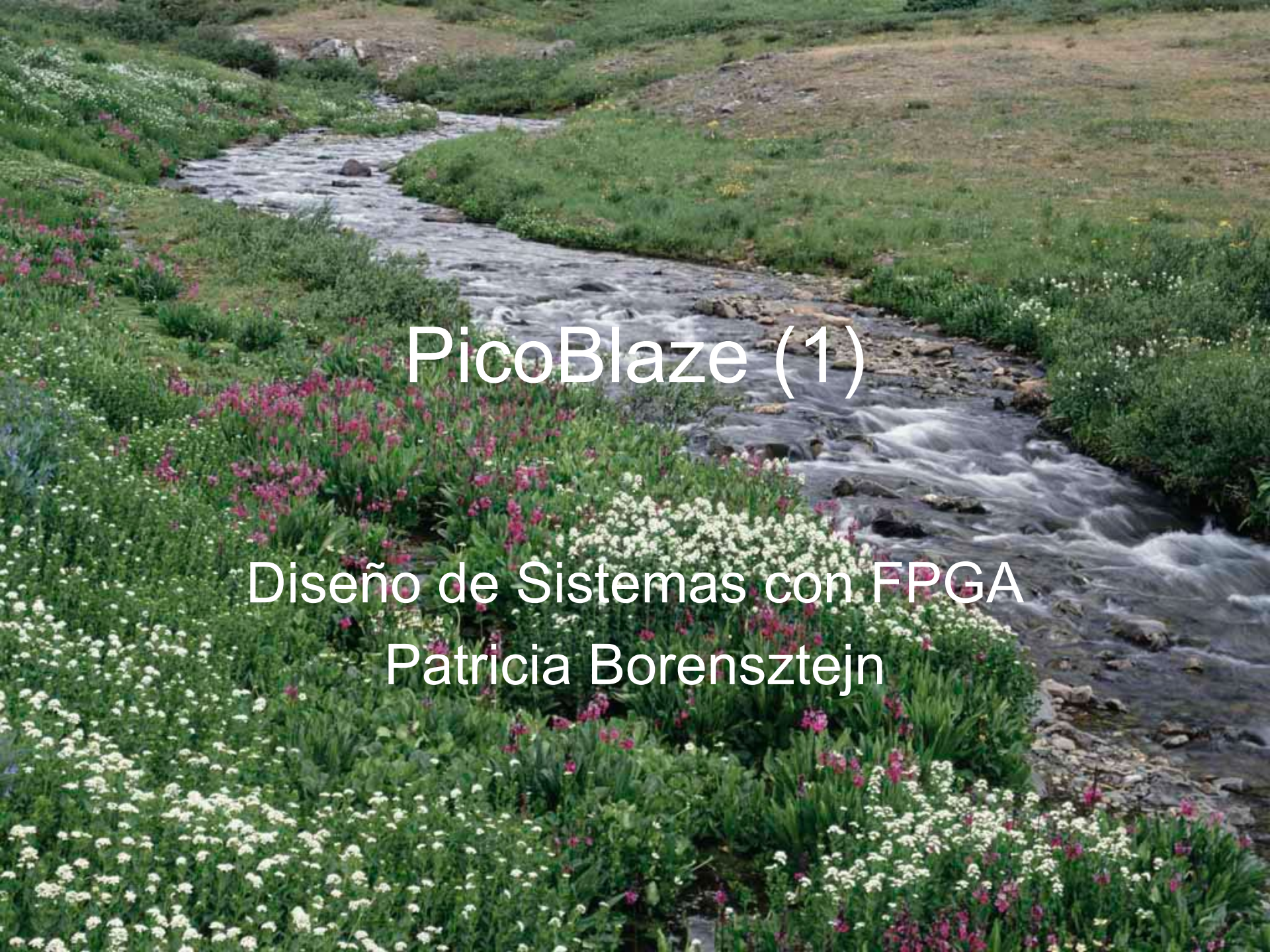


PicoBlaze (1)

Diseño de Sistemas con FPGA

Patricia Borensztejn

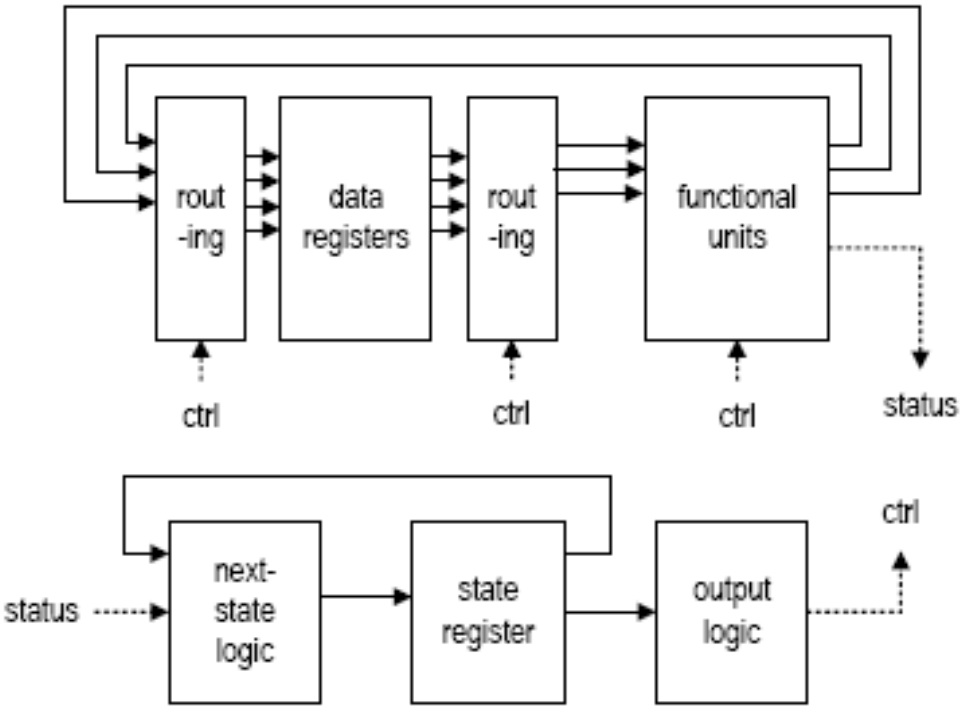
PicoBlaze

- Microcontrolador de 8 bits.
- Nucleo de procesador desarrollado para los FPGA de xilinx
- Se provee como soft core, y puede sintetizarse con el resto de la lógica.
- Está optimizado y ocupa solo 200 celdas lógicas. (5% de los recursos del 3S200)

De un FSMD a un microprocesador

- La metodología de diseño FSMD y la descripción a nivel RTL permite transformar un algoritmo en hardware.
- En un FSMD todos los componentes, el número de registros, la conexión entre ellos (el ruteo), los tipos de UF´s, etc, están hechos a la medida exacta de la aplicación.
- Una alternativa es utilizar el hardware para correr distintas aplicaciones en función del “programa” (software).

FSMD



(a) Block diagram of an FSMD

De un FSMD a un microprocesador

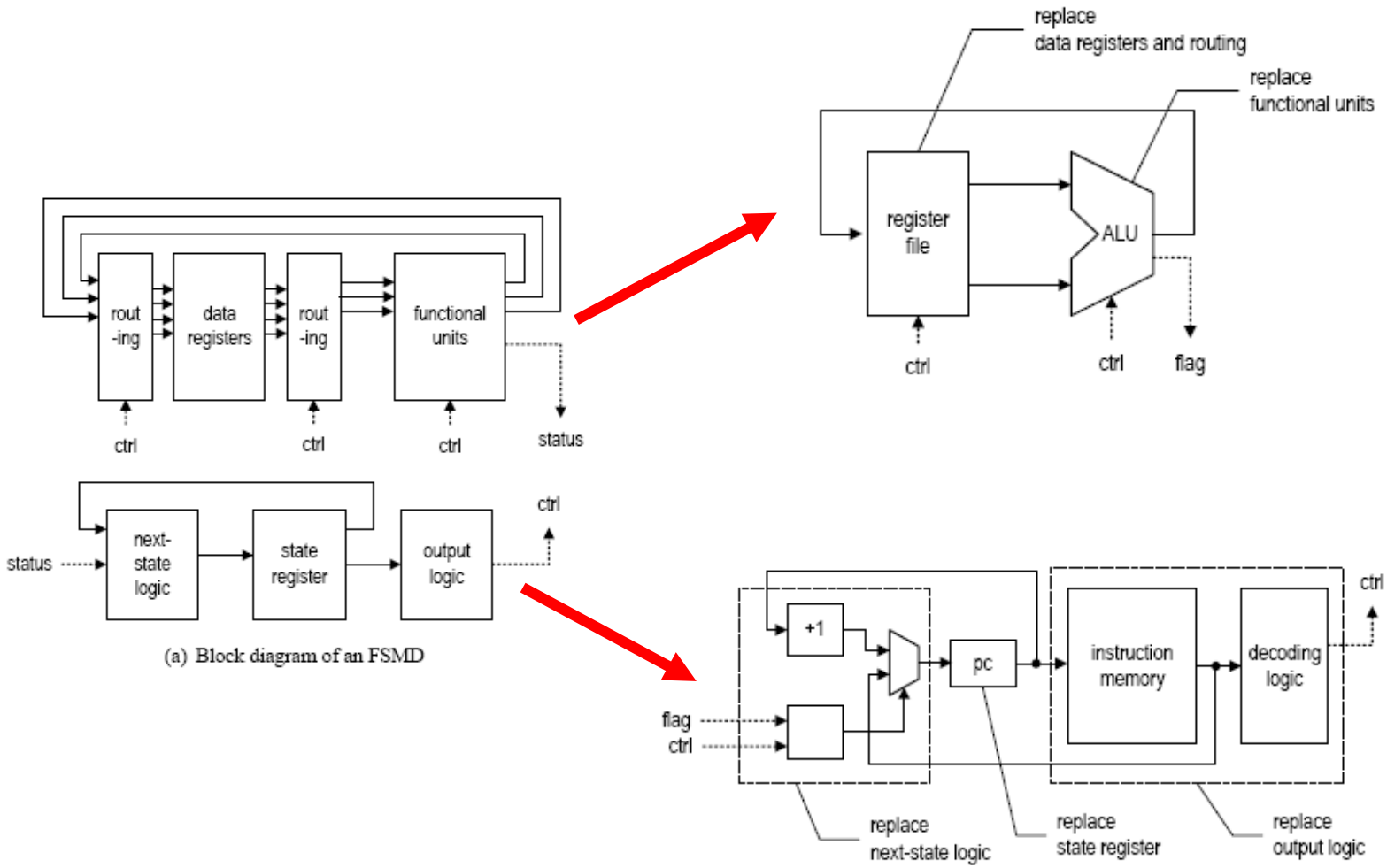
- Se reemplaza el Data Path por una configuración fija:
  - Los registros con sus ruteos específicos se reemplazan por un Banco de Registros con un números fijo de registros y dos puertos de lectura y uno de escritura.
  - Las unidades funcionales se reemplazan por una ALU que puede realizar un conjunto predefinido de operaciones (aritméticas y lógicas). Dichas operaciones se describen en el formato RTL:
    - rd<= r1 op r2

    donde r1 y r2 son las direcciones de dos registros fuentes y rd la dirección de un registro destino y op es una de las funciones disponibles en la ALU

De un FSMD a un microprocesador

- Se reemplaza la FSM con una máquina de estados “programable”, reemplazando:
  - El registro de estado de la FSM por un contador de programa (PC). El contenido del PC representa el estado actual del camino de control
  - Las señales de salida asociadas a cada estado en la FSM se reemplazan por “instrucciones” que se almacenan en un módulo de memoria (memoria de instrucciones) . La dirección de memoria corresponde a un estado (del PC). La instrucción apuntada por el PC se obtiene y se decodifica para generar las señales de salida. La memoria de instrucciones y la lógica de decodificación funcionan como un circuito sofisticado para las funciones de salida
  - En un FSM, el siguiente estado depende de las entradas y puede ser cualquiera. En un FSM programable el siguiente estado es usualmente, el estado actual mas uno (PC=PC+1) lo cual refleja la naturaleza secuencial de ejecución. Esta solo puede ser alterada por instrucciones especiales (jump) donde el PC se carga con un valor diferente. El incrementador y la logica de multiplexación para el PC funciona como el circuito siguiente estado.

Diseño de un microcontrolador



De un FSMD a un microprocesador

- Una vez que hemos reemplazado el Data Path por la ALU y el Banco de Registros, y la máquina de estados FSM por una Máquina de estados Programable, para hacer el sistema específico es necesario desarrollar la secuencia de instrucciones específicas y cargarlas en la memoria.
- De esta forma, el FSMD es el mismo para muchas aplicaciones, habiéndose transformado en una PLATAFORMA DE PROPOSITO GENERAL.
- Esta plataforma constituye la base del nucleo de Picoblaze.

PicoBlaze

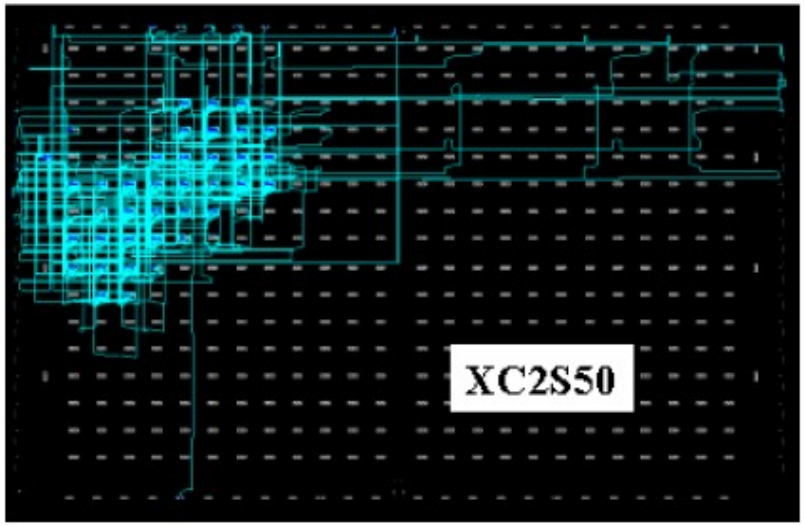
- A diferencia de FSMD que podría completar en un único estado (ciclo de reloj) una operación compleja, el PicoBlaze puede realizar una operación prediseñada en un ciclo. Puede requerir muchas instrucciones para realizar la misma tarea que el FSMD.
- ¿Cuando es mejor usar microcontrolador a desarrollar hardware?
  - Para operaciones no críticas en tiempo. Por ejemplo, para interfase con E/S.

Diseñando PicoBlaze

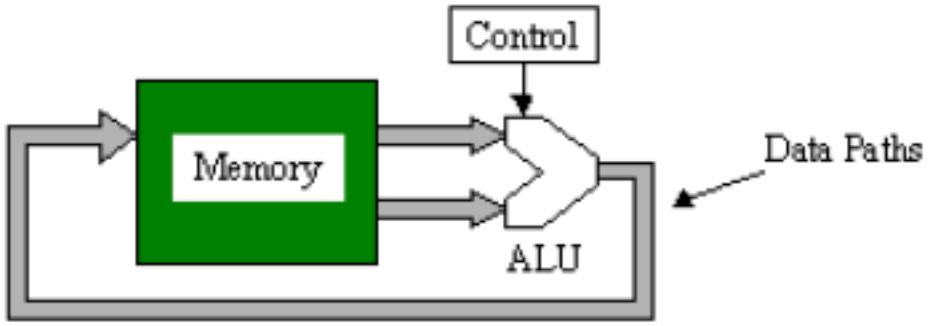
- A continuación, vamos a seguir el proceso de diseño de PicoBlaze, realizado por Ken Chapman, ingeniero de Xilinx.
- Seguiremos el paper:
  - “Creating Embedded Microcontrollers”, Ken Chapman.
  - (para los amantes de arquitectura…. Una joyita!). Enjoy…..

Introducción

- Se trata de diseñar un procesador pequeño, al cual se refiere como PSM: “Programmable State Machine”. También los llama : “very small processor macros”.
- Mas cercanos al mundo de los microcontroladores
- Debe ocupar muy poca área
- Util para aplicaciones con complejas máquinas de estado y bajo requerimiento de velocidad.
- En Spartan-II usa 36 CLB´s. Si pones 8 de estos aún te queda libre el 30% del área.
- El los llama: KCPSM (Ken Chapman PSM)



¿Cuantos Bits?



- ¡Los que tu necesites!
- Pero… cuantos mas bits, mas lógica (aunque mas poderoso el procesador)
- Cuantos menos bits, mas pequeña la implementación pero mas bajo el rendimiento aritmético.
- El primer diseño lo hizo dejando la definición del ancho de los datos para el momento de la síntesis. Eso complicó un poco el diseño.
- Luego le preguntó a los usuarios cuantos bits habían usado…. 8 dijeron! Y así fue.
- Sin embargo… dice K.C “ consideren tamaños no standards si eso es lo que necesitan!!!!

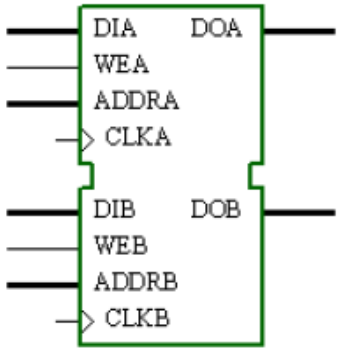
Programa Embebido

- Un procesador standard obtiene sus instrucciones de una memoria externa.
- En este caso, el PSM debía estar totalmente embebido dentro de la FPGA y ser totalmente autocontenido, de forma que uno pudiera poner tantos de esos como quisiera, sin depender del diseño del PCB donde estuviera la FPGA.
- O sea que la memoria tenía que implementarse dentro del dispositivo (FPGA).
  - Antes no había bloques de memoria embebidos en la FPGA, por lo tanto estos debían realizarse con CLB´s, por lo tanto los programas debían ser muy sencillos
  - Virtex introdujo los bloques de RAM…Asi que estos bloques se conviertieron en la ROM de los programas.
  - Entonces: el tamaño de los programas esta limitado por un bloque de RAM embebido.

Block RAM Aspect Ratios

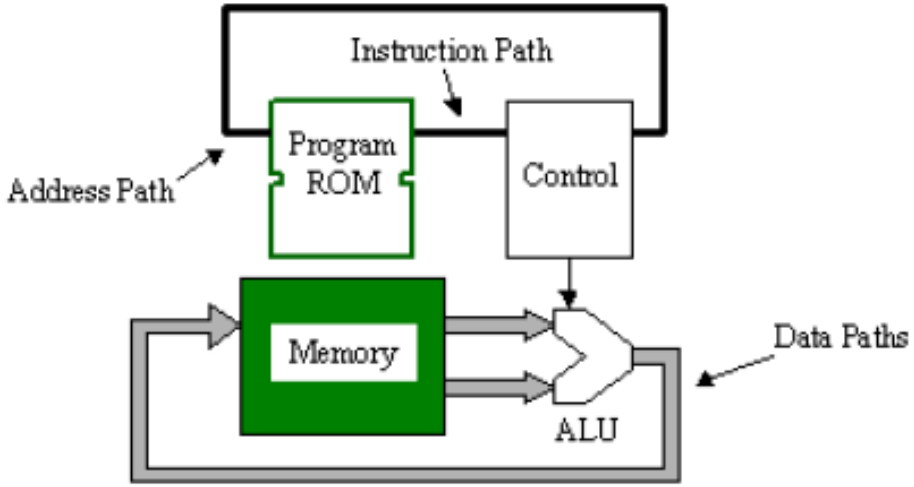
Locations	Instruction Width	Address Bits
4096	1	12
2048	2	11
1024	4	10
512	8	9
256	16	8

Block RAM



Programa Embebido

- Una ventaja del programa embebido es que todas las entradas y salidas del procesador son “pines virtuales” dentro de la FPGA.
- Esto implica que no es necesario usar buses de tiempo compartido para datos e instrucciones ➔ es obvio utilizar arquitectura Harvard!
- La arquitectura Harvard, memorias separadas para datos e instrucciones, implica también que el tamaño de los datos no tiene porqué coincidir con el tamaño de las instrucciones.

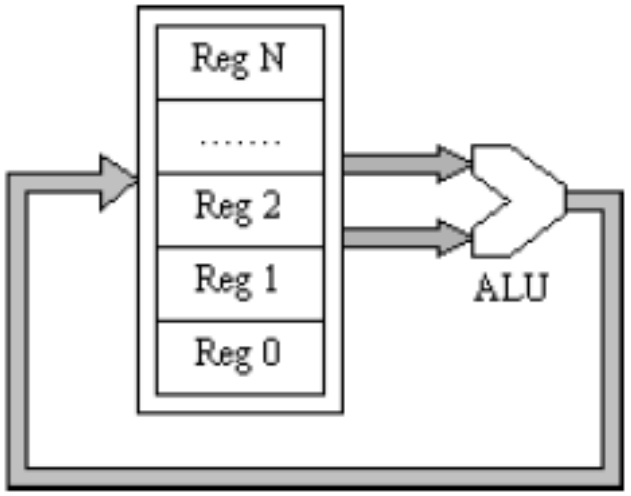


Tamaño de instrucción

- ¿Cual sería entonces el tamaño ideal para las instrucciones?
  - Si es estrecho:
    - Ocupa menos posiciones de memoria
    - Necesitamos mas lógica de decodificación
    - Probablemente los operandos que formen la instrucción no puedan estar incluidos en la misma instrucción o sea que necesitamos mas ciclos de fetch➔esto aumenta la complejidad de la lógica de secuenciamiento y produce pérdida de rendimiento.
  - Si es mas ancha:
    - Cuantos mas bits dedicamos a la instrucción, menos lógica de decodificación necesitamos
    - Cuantos mas bits dedicamos a la instrucción, menos instrucciones caben en la memoria ROM. Entonces los programas tienen que ser mas cortitos.
    - Cuantos mas bits dedicamos a la instrucción, es probable que no haya necesidad de ir a buscar operandos inmediatos.
- KC optó por:
  - Tamaño de instrucción de 16 bits. Utilizando un BRAM de 256 posiciones.

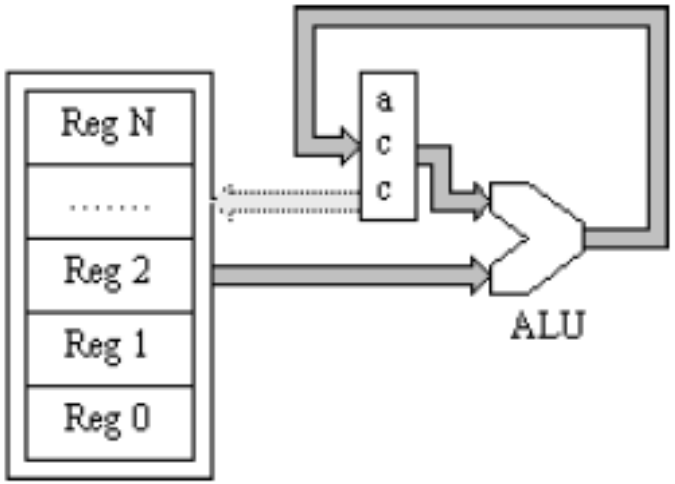
¿Registros, Acumulador o Pila?

- Hay 3 arquitecturas básicas para obtener los operandos.
  - Arquitectura registro-registro:
    - Usa un conjunto de registros. Mas sencilla la programación. Compleja implementación debido a los multiplexores necesarios para seleccionar el registro.
    - Tiene impacto sobre el formato de instrucción ya que la identificación del registro debe ir en la instrucción.



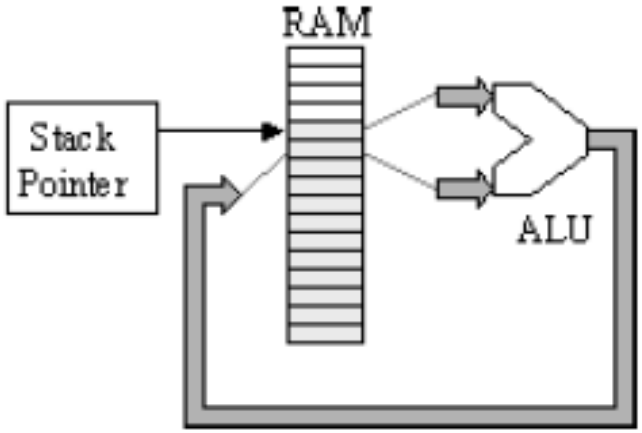
¿Registros, Acumulador o Pila?

- Arquitectura de Acumulador:
  - Usa memoria o registro para el acumulador
  - Uno de los operandos y el destino es implícito por lo tanto no necesita codificarse en la instrucción
  - El programa debe ocuparse de mover del/al acumulador. Los programas pueden volverse mas lentos porque estamos siempre inicializando o cargando el acumulador



¿Registros, Acumulador o Pila?

- Arquitectura de Pila:
  - Es la más eficiente en la implementación. No necesita selección mas que la del SP.
  - La instrucción no necesita bits para codificar la ubicación de los operandos.
  - Se necesitan mas instrucciones PUSH y POP, para asegurarse del contenido del TOP del stack
  - Los programas mejoran si se escribe en notación polaca! Eso desalienta a la gente.

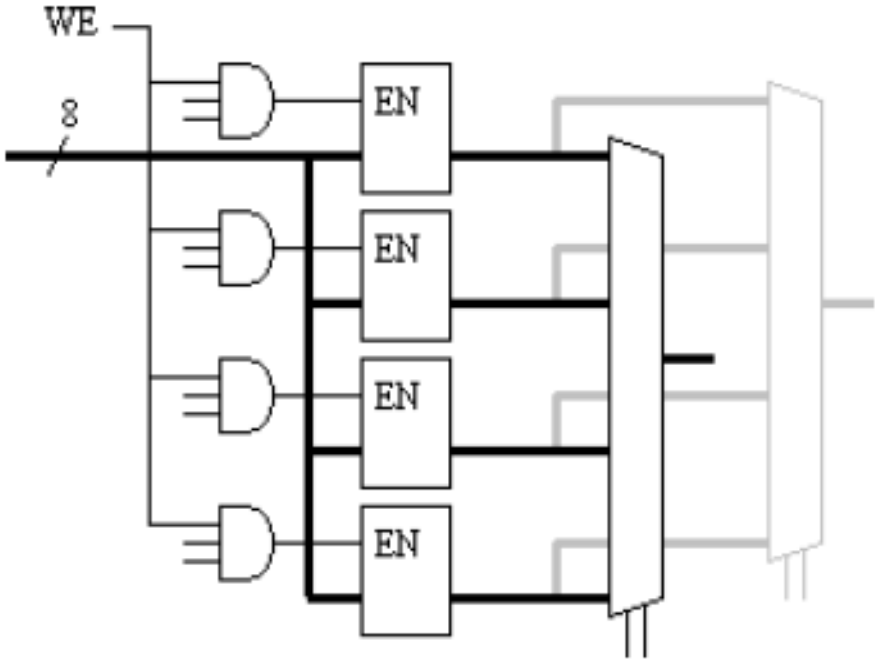


¿Registros, Acumulador o Pila?

- Comparación:
  - La mas óptima es la de pila. Pero la metodología de programación es pesada.
  - La de acumulador es razonable, pero usa muchas instrucciones para mover datos del/al acumulador y no tenemos mucho espacio para la memoria de instrucciones.
  - ¡La de registro me gusta mucho!! Pero es cara… a ver si podemos con ella…

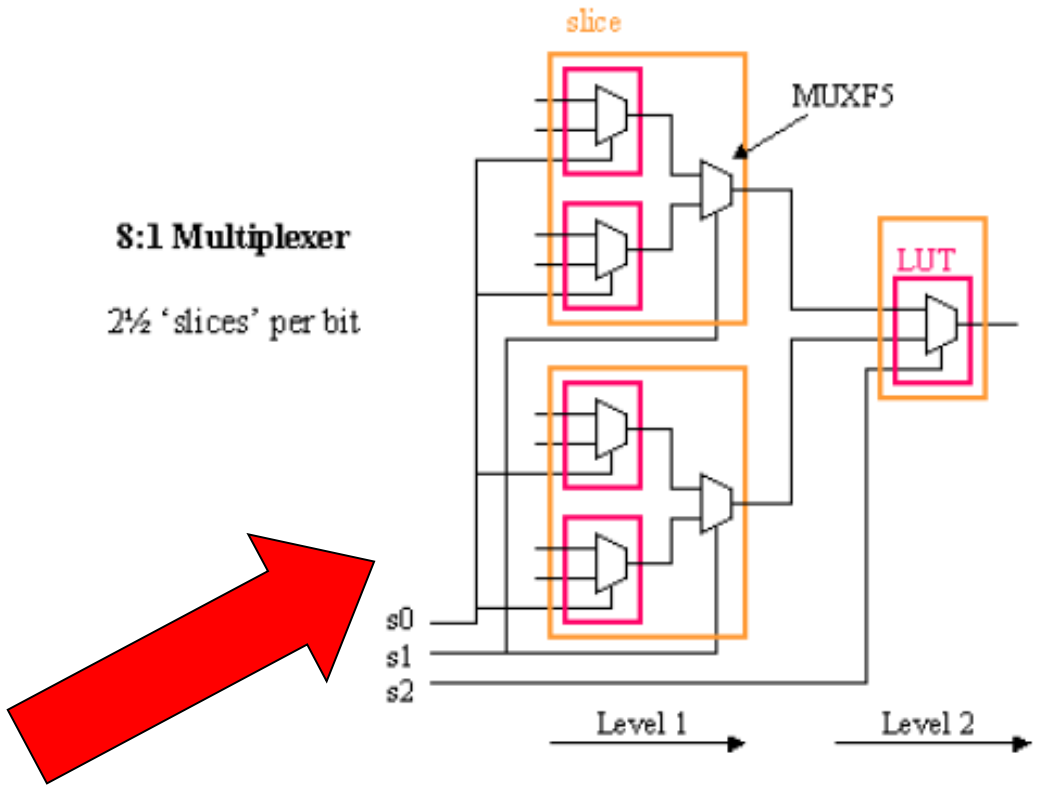
Implementando la arquitectura registro-registro

- Supongamos que tenemos 4 registros de 8 bits. Necesitamos 32 flip-flops, o sea, 16 “slices”.
- Necesitamos clock-enables para los registros. La lógica para el EN requiere 2 slices en total. Es una función AND de 3 entradas y se implementa con las LUTs (1 EN por registro, 4 LC, 2 slices)



Implementando la arquitectura registro-registro

- Además, se necesita multiplexores para seleccionar los operandos (dos multiplexores, 4 a 1).
- Un multiplexor 4 a 1 usa un slice completo por bit combinando las dos LUTs y MUXF5. Este multiplexor de 8 bits necesita 8 slices. Por lo tanto dos multiplexores, 16 slices.
- (Si en lugar de 4 registros tuviéramos 8 registros, necesitaríamos un multiplexor de 8:1, pero para hacer esto, necesitamos otro nivel de lógica!)
- Observación: un multiplexor de 2 a 1 es una función de 3 entradas.



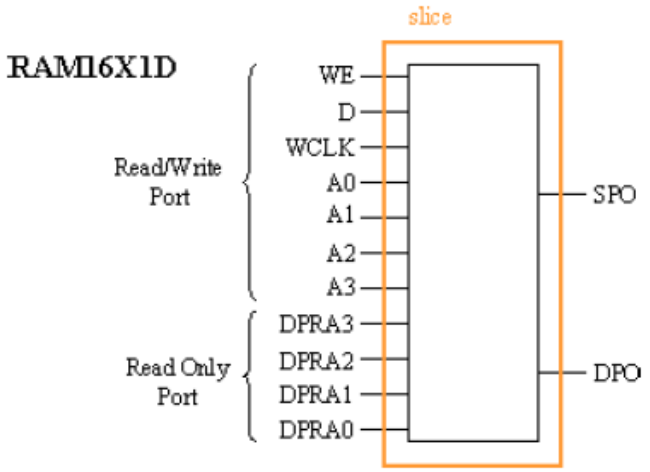
Implementando la arquitectura registro-registro

8-bit Register Bank Resources (all figures in 'slices')

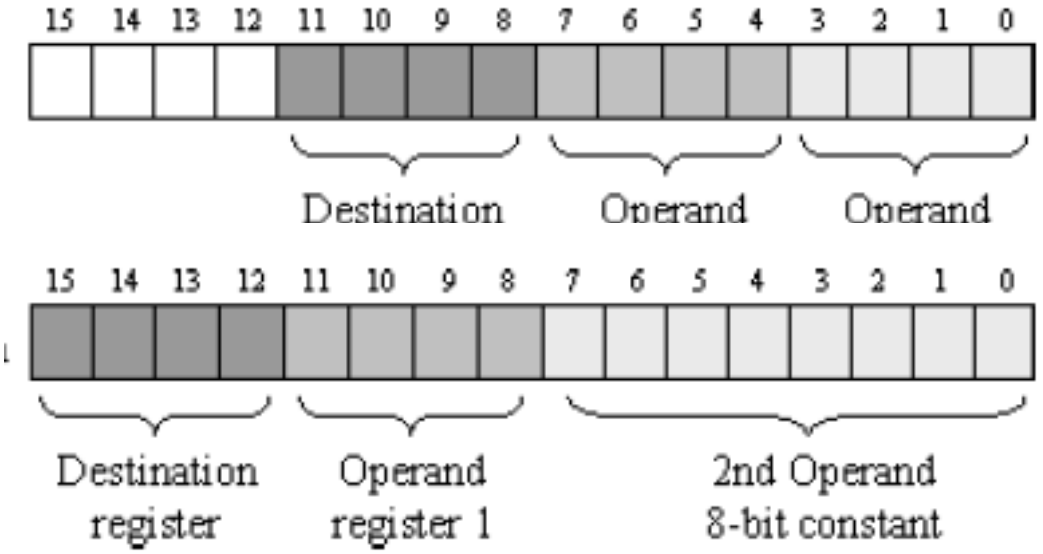
№. Registers	Flip Flops	1-bit Mux	Dual register select Mux's	Enable gates	Packed Size
4	16	1	16	2	18
8	32	2½	40	4	44
16	64	5	80	8	88

Implementando la arquitectura registro-registro

- Y entonces apareció en juego la característica que ofrece xilinx de poder implementar SRAM con LUT´s.
- Cada LUT ofrece 16 bits de RAM, cada Slice 32.
- En un Slice, podemos tener 16 bits de RAM configurada como de doble puerto lo cual es idóneo para implementar un Banco de Registros.
- Es decir, se puede implementar un Banco de 16 Registros de 8 bits utilizando un total de 8 slices (contra 88 usando flip-flops)

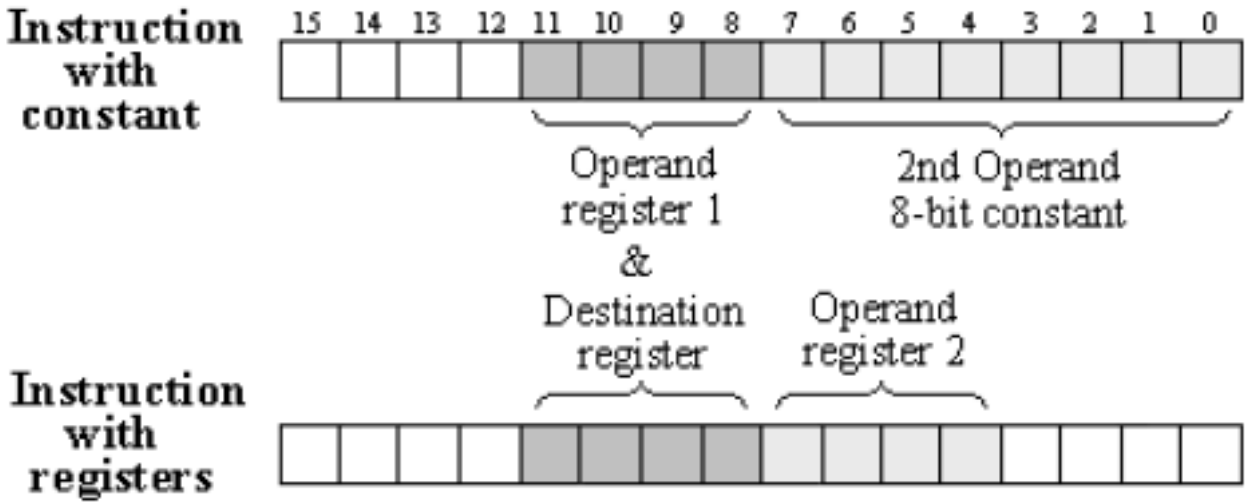


Instrucciones: Formatos



- Definida la arquitectura registro-registro podemos definir el formato de instrucción que debe contener campos para identificar cada uno de los 16 registros.
- Pero además queremos operaciones que envuelvan valores constantes (de 8 bits). Pero ahora no cabe el código de operación!
  - Una posibilidad es aprovechar el doble puerto de la BRAM trayendo 32 bits de instrucción.
  - Otra posibilidad: hacer el operando destino el mismo que el fuente, liberando 4 bits!
  - Otra posibilidad: hacer la instrucción mas grande… por ejemplo de 18bits aprovechando los dos bits de paridad.

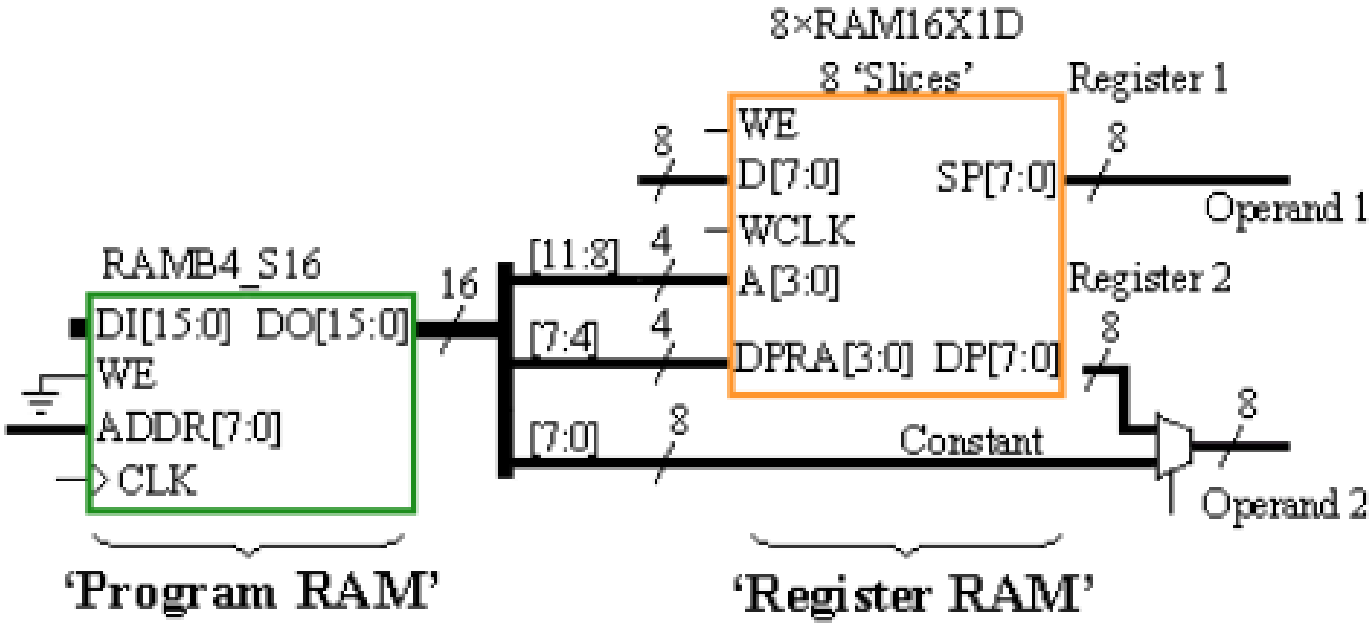
Instrucciones: Formatos



- Igual que Intel

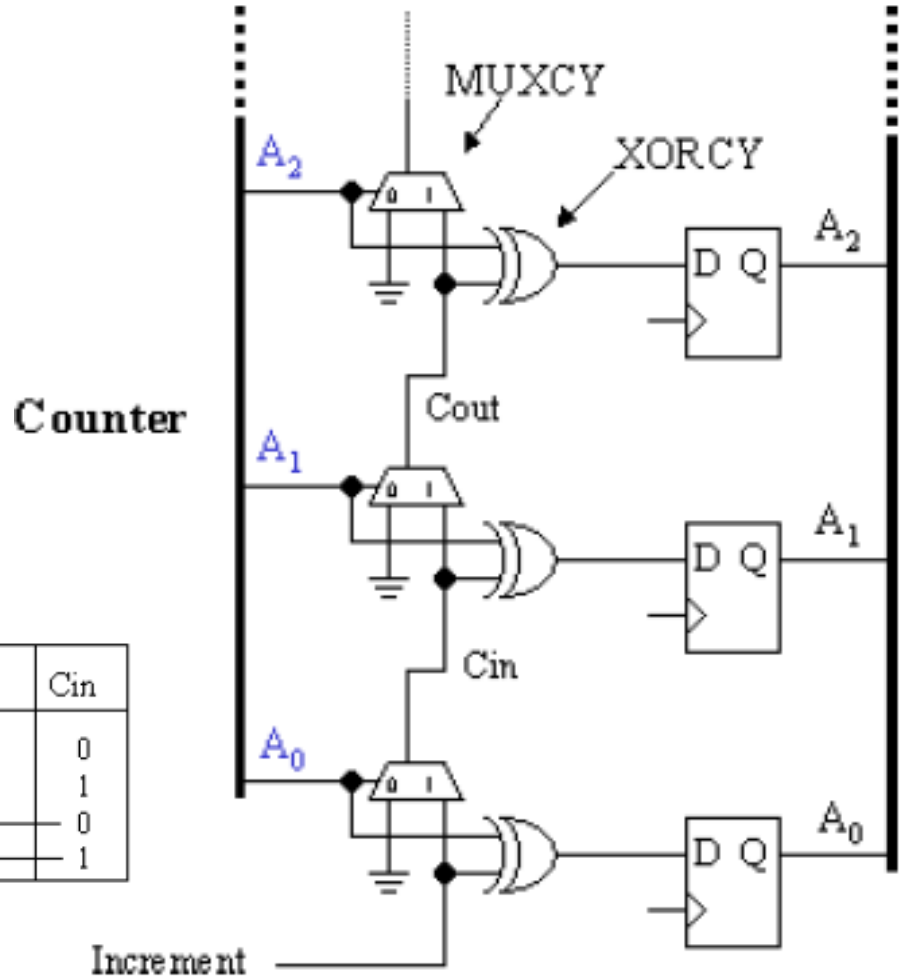
Memoria controlando Memoria

- Esto es lo que tenemos hasta ahora:
  - Un programa en memoria que actúa como control y está formado por BRAM
  - Los datos guardados en registros, que están implementados mediante memoria distribuida.



Los programas son secuenciales casi todo el tiempo

- La operación de incrementar el PC se realiza de forma muy eficiente usando la cadena de acarreo, sin usar la LUT.
- Para hacer PC<= PC+1 siendo el PC de 8 bits, se requieren 4 slices.



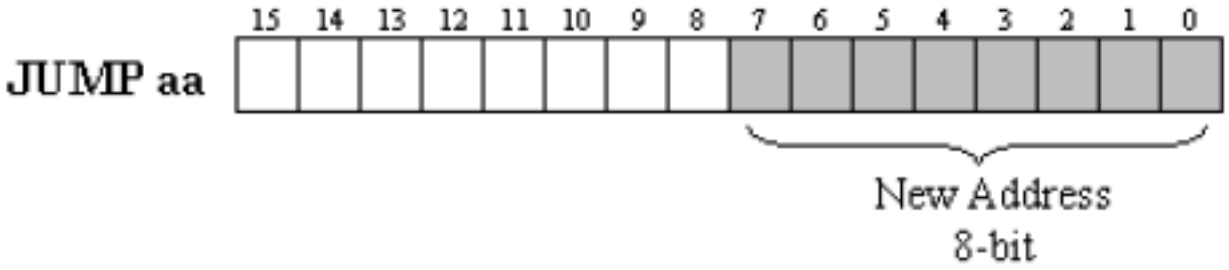
XORCY

A_n	Cin	A_{n+1}
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

MUXCY

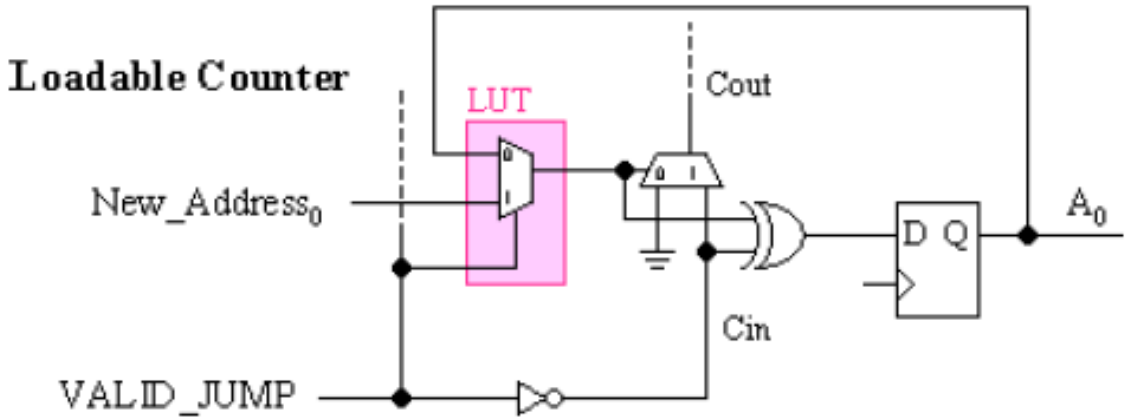
A_n	GND	Cout	Cin
0	0 →	**0**	0
0	0 →	**0**	1
1	0	**0** ←	0
1	0	**1** ←	1

JUMP



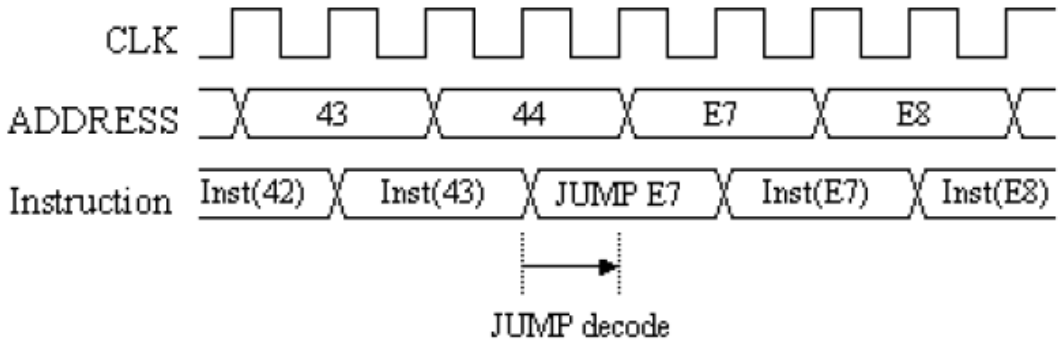
- Es necesario que el PC se cargue a veces con el valor incrementado en uno, y otras veces con el valor de 8 bits almacenado en la instrucción de JUMP.
- Se utiliza la LUT para implementar el multiplexor.

- La señal VALID_JUMP se niega para evitar que se incremente la nueva dirección aa mientras es cargada.



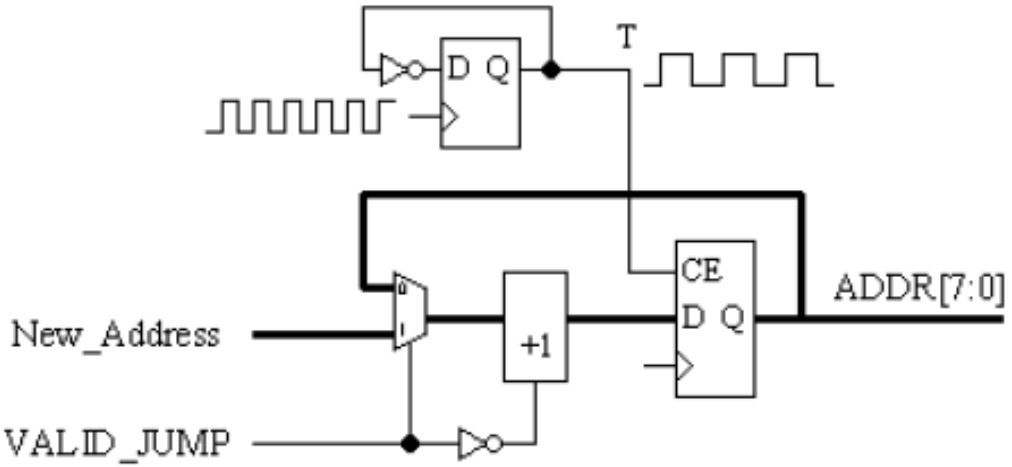
Dos ciclos por instrucción

- La instrucción JUMP necesita un ciclo para ser decodificada y así determinar la dirección de la siguiente instrucción.
- Una vez que se ha determinado la dirección de la siguiente instrucción, la RAM necesita un ciclo para accederla, entonces, una instrucción necesita dos ciclos para ejecutarse.



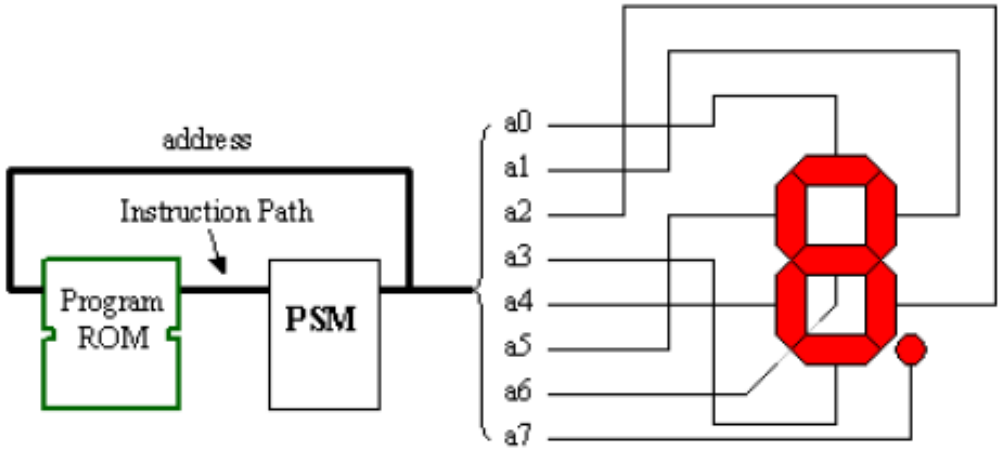
Dos ciclos por instrucción

- Las demás instrucciones no necesitan un ciclo extra para determinar el PC siguiente, por lo tanto, pueden ejecutarse en un ciclo. Sin embargo, para hacer predecible el funcionamiento del PSM vamos a hacer que todas las instrucciones tarden dos ciclos en completarse.
- Es necesario entonces una lógica de retardo para hacer que el PC cambie cada dos ciclos. Esto se hace controlando el EN de todos los flip.flops, con un flip-flop.



Primer Programa

- Sin necesidad de usar la ALU, y, aprovechando el hecho de que todas las salidas del procesador están disponibles en la FPGA, se controla el siete segmentos!
- El display cambia cada dos ciclos.



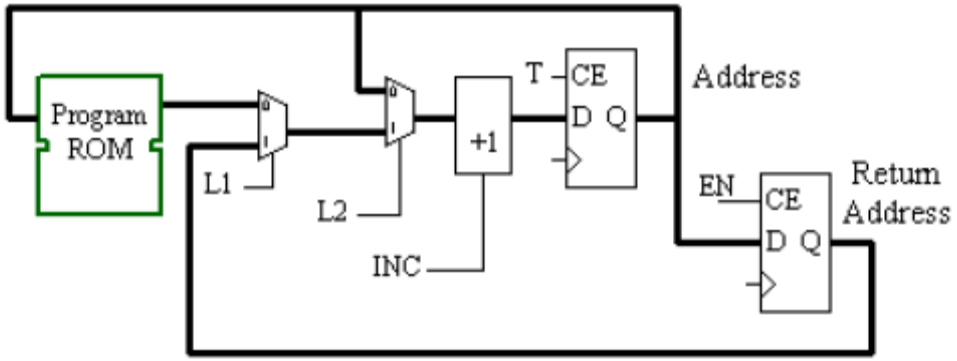

```
Addr Instruction    Comment

 03    JUMP 5B      ; display '1' and go to '2'
 07    JUMP 7F      ; display '7' and go to '8'
 3F    JUMP 03      ; display '0' and go to '1'
 4F    JUMP 66      ; display '3' and go to '4'
 5B    JUMP 4F      ; display '2' and go to '3'
 66    JUMP 6D      ; display '4' and go to '5'
 6D    JUMP 7D      ; display '5' and go to '6'
 6F    JUMP 3F      ; display '9' and go to '0'
 7D    JUMP 07      ; display '6' and go to '7'
 7F    JUMP 6F      ; display '8' and go to '9'
```

Subrutinas

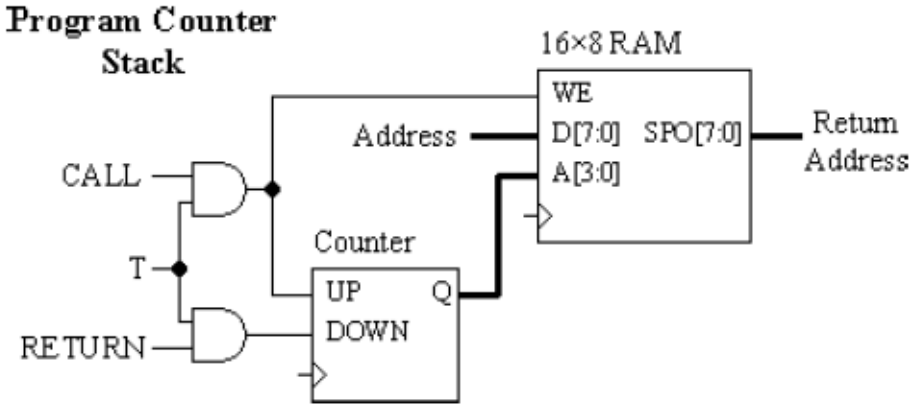
- La instrucción CALL es idéntica a la de JUMP salvo que además hay que salvar el PC actual.
- El valor del PC se guardará en registros especiales para poder luego ser utilizado por la instrucción Return.
- El PC entonces puede provenir de 3 fuentes:
  - PC+1
  - AA de la instrucción JUMP o CALL
  - Registro de direcciones de retorno

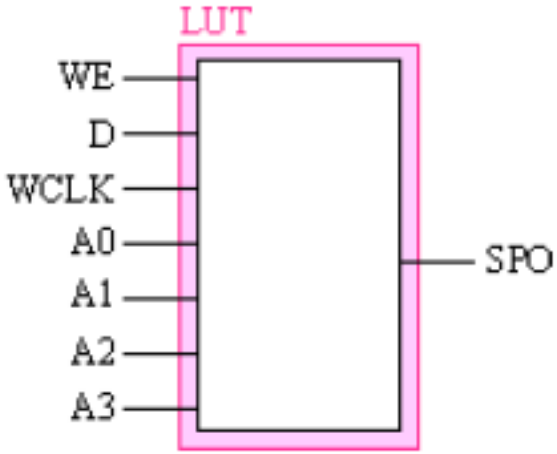
Instruction	EN	L1	L2	INC
Normal	0	×	0	1
JUMP	0	0	1	0
CALL	1	0	1	0
RETURN	0	1	1	1



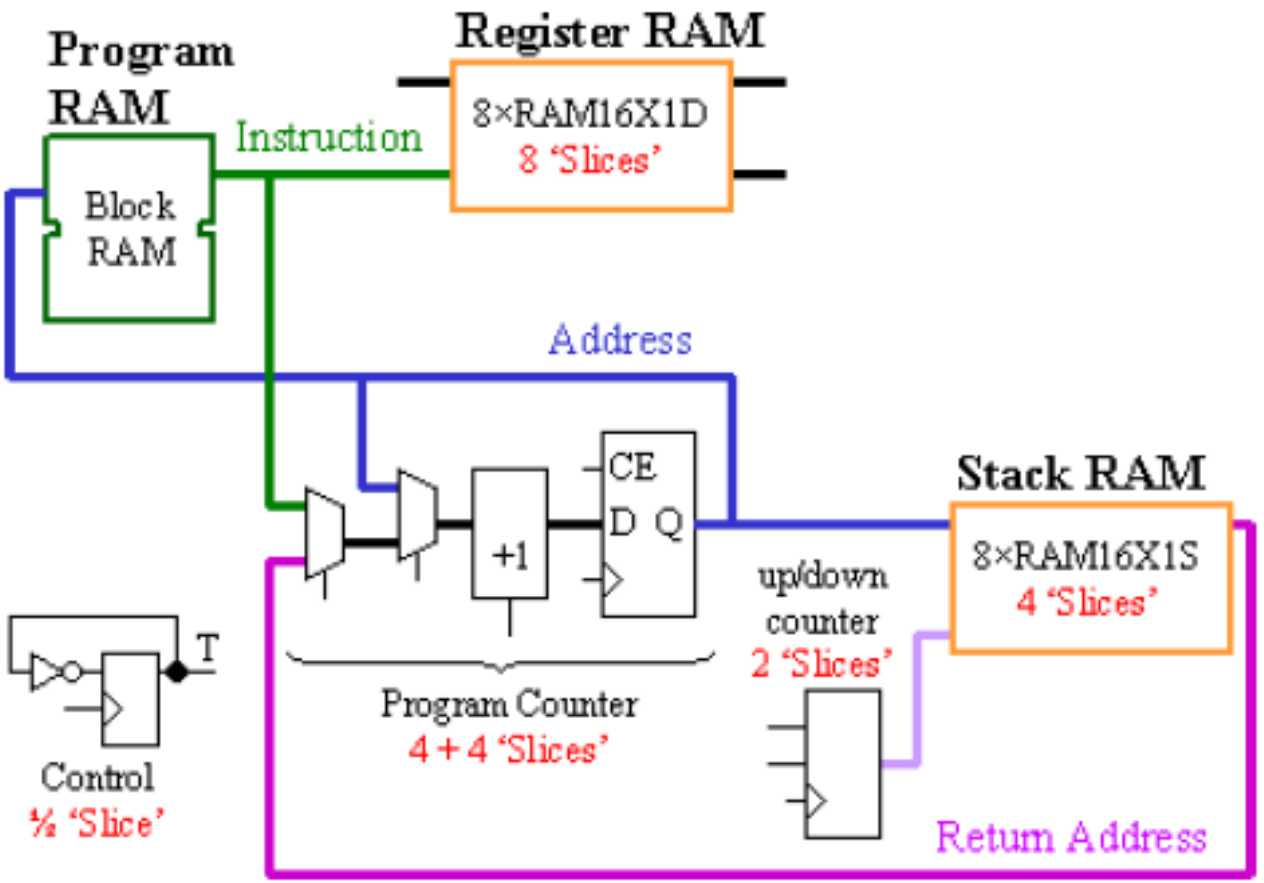
Registros para Dirección de Retorno

- Se realiza una pila de direcciones de retorno utilizando RAM distribuida, de forma que se puedan anidar subrutinas.
- Para hacer la pila, se utiliza un contador de 4 bits hacia arriba o hacia abajo (pila de 16 registros de 8 bits). EL CALL incrementa el contador. El return lo decrementa. El contador necesita 2 slices (4 bits).





RAM, RAM y más RAM



Cuentas

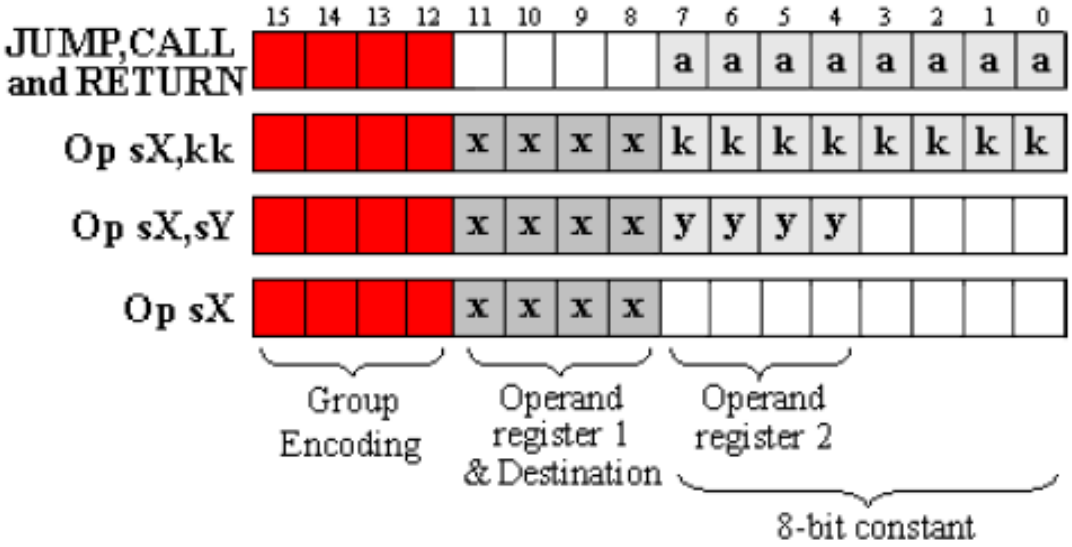
- Se ha usado hasta ahora:
  - 12 slices de memoria distribuida para implementar el Banco de Registros y la Pila de direcciones de retorno
  - 4 slices para PC+1
  - 4 slices para dirección del JUMP (una LUT por bit)
  - Un retardo de control: medio slice
  - Contador para la pila: 2 slices
- Si no se hubiera usado memoria distribuida hubieramos necesitado: 152 slices (vs 12)

Conjunto de Instrucciones

- Definir el conjunto de instrucciones depende de lo que a uno le interese hacer.
- Dentro de la FPGA, uno podria definir un conjunto diferente y apropiado para cada aplicación.
- Pero, si la intención es armar una máquina de propósito general que sea útil para distintas aplicaciones, el foco debe estar en elegir aquellas operaciones que mantengan el tamaño pequeño de la máquina.
- Por otro lado, recordemos que tenemos poco lugar para instrucciones, por lo tanto el código debe ser pequeño y eso nos llevaría quizas a incluir operaciones mas contundentes (multiplicación?)

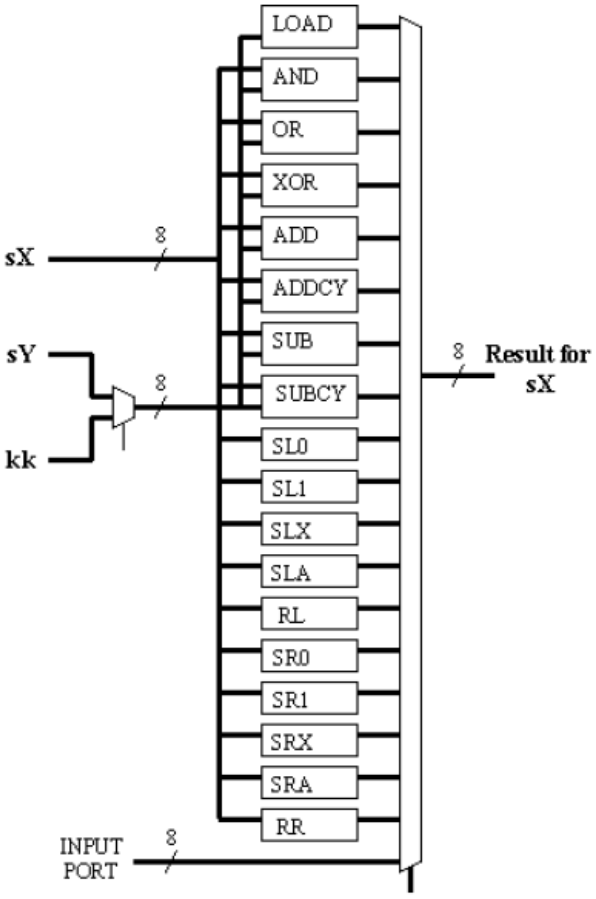
Conjunto de Instrucciones: Restricciones

- Cada uno de los formatos que tenemos dejan mas de 4 bits para la operación. La mas restrictiva es Op sk,kk ya que no deja más que 4 bits.
- La idea es que 4 bits sirvan para identificar los grupos principales. Dentro de cada grupo, se usan bits dispersos para identificar la operación.
- En el caso del formato Op sk,kk tendremos que codificar la operación en los 4 bits. Pero no solo eso. Esos 4 bits nos deben servir para identificar a los otros grupos (los 3 restantes). Por lo tanto, el máximo número de intrucciones en ese grupo es 13.



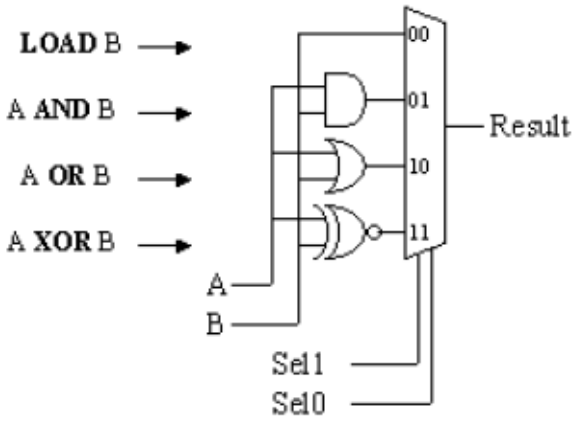
Lógica para Multiplexar

- Supongamos esta primitiva implementación de una ALU: todas las operaciones se realizan en paralelo, y según la decodificación de la instrucción se selecciona el resultado.
- Para las 18 operaciones aritméticas definidas en PicoBlaze, necesitariamos un multiplexor de 18 resultados mas uno para el puerto de entrada y otro pequeño de 2 a 1 para seleccionar el segundo operando.
- Implementar este multiplexor para datos de 8 bits nos ocupa 48 slices.
- Otra complejidad son las líneas de control del multiplexor que provienen de bits dispersos de la instrucción.
- Se puede reducir la complejidad, reduciendo el número de operaciones…. Pero no es lo que queremos!!!



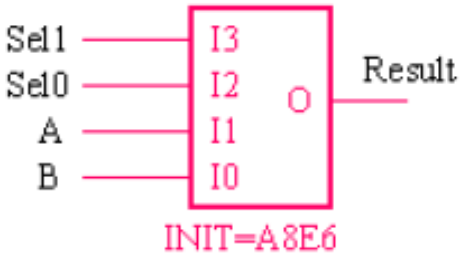
Grupo Lógico

- Dos maneras de verlo:



- Ocupa 4 slices por cada AND, OR, XOR mas 8 slices para el multiplexor 4:1. Total 20 slices

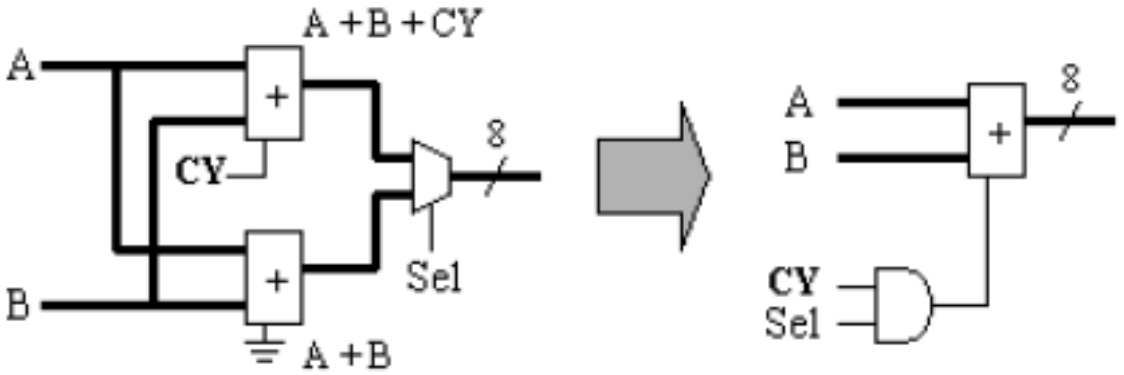
Logical Group					
I3	I2	I1	I0	O	INIT=A8E6
Sel1	Sel0	A	B	Result	
0	0	0	0	0	6
0	0	0	1	1	
0	0	1	0	1	
0	0	1	1	0	
0	1	0	0	0	E
0	1	0	1	1	
0	1	1	0	1	
0	1	1	1	1	
1	0	0	0	0	8
1	0	0	1	0	
1	0	1	0	0	
1	0	1	1	1	
1	1	0	0	0	A
1	1	0	1	1	
1	1	1	0	0	
1	1	1	1	1	



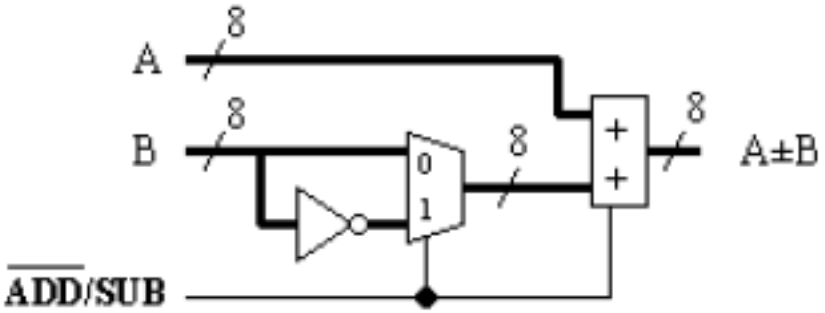
- Es una LUT por bit ! Ocupa 4 slices.

Grupo Aritmético

- ADD y ADDC
  - 4 slices por cada ADD o SUB, mas 8 de multiplexar 4:1. Total 24 slices

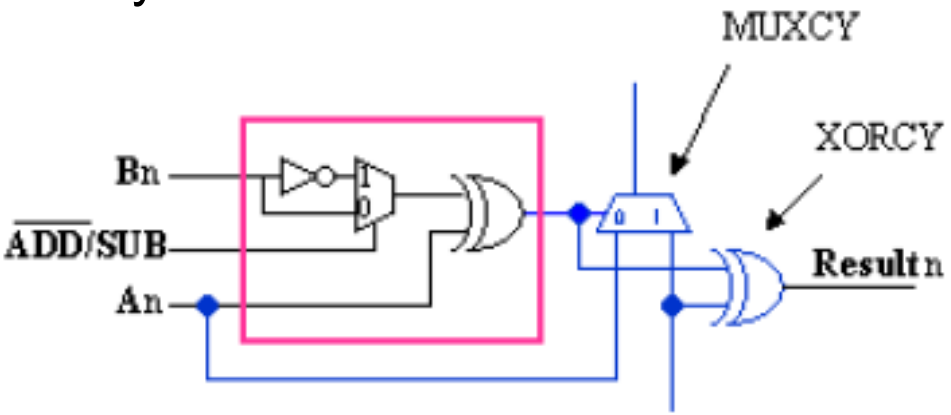


- SUB y ADD



Grupo Aritmético

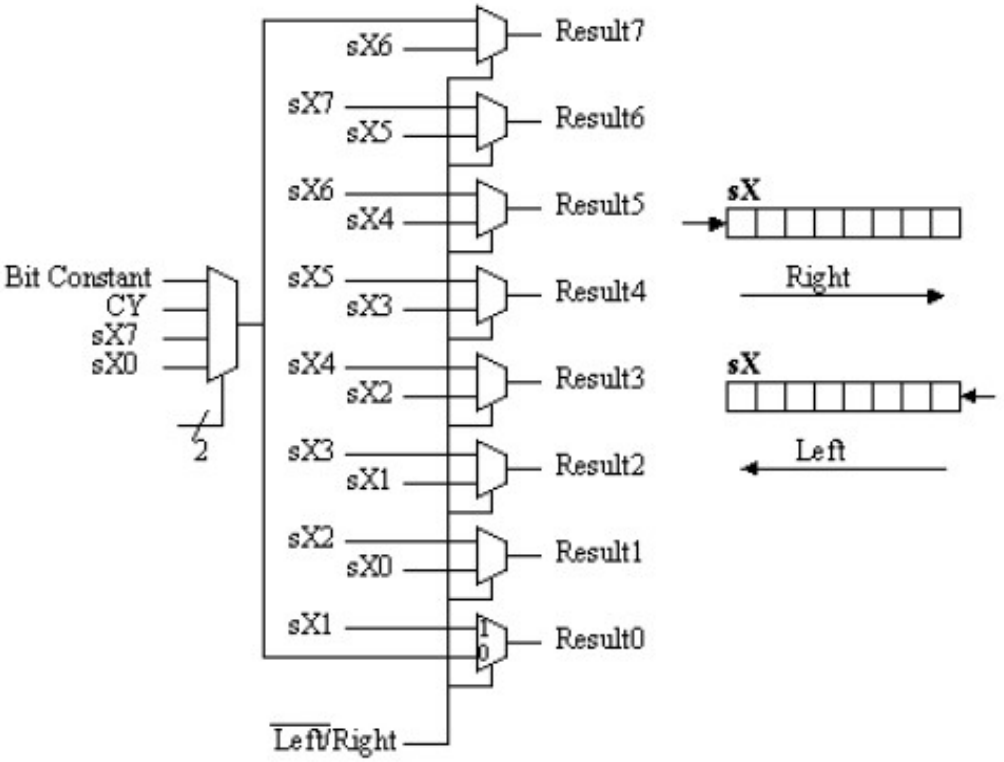
- Se usan 4 slices y medio. El medio es para la puerta AND de Sel y CY.



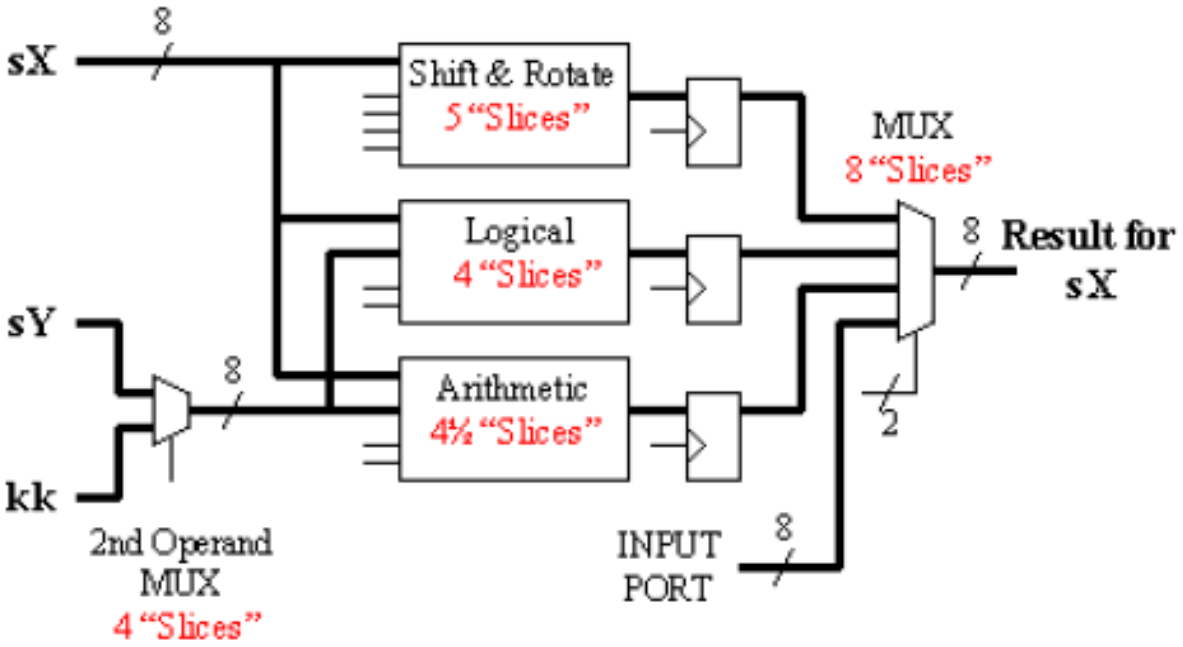
Instruction	$\overline{\text{ADD}}$/SUB	Include CARRY	Carry Chain Cin
ADD	0	0	0
ADDCY	0	1	CY
SUB	1	0	1
SUBCY	1	1	not CY

Shift y Rotate

- Se usan 4 slices (8 multiplexores 2:1).
- Un slice mas para seleccionar el bit adicional.
- Las líneas de control de los multiplexores salen de los bits dispersos del codigo de operación

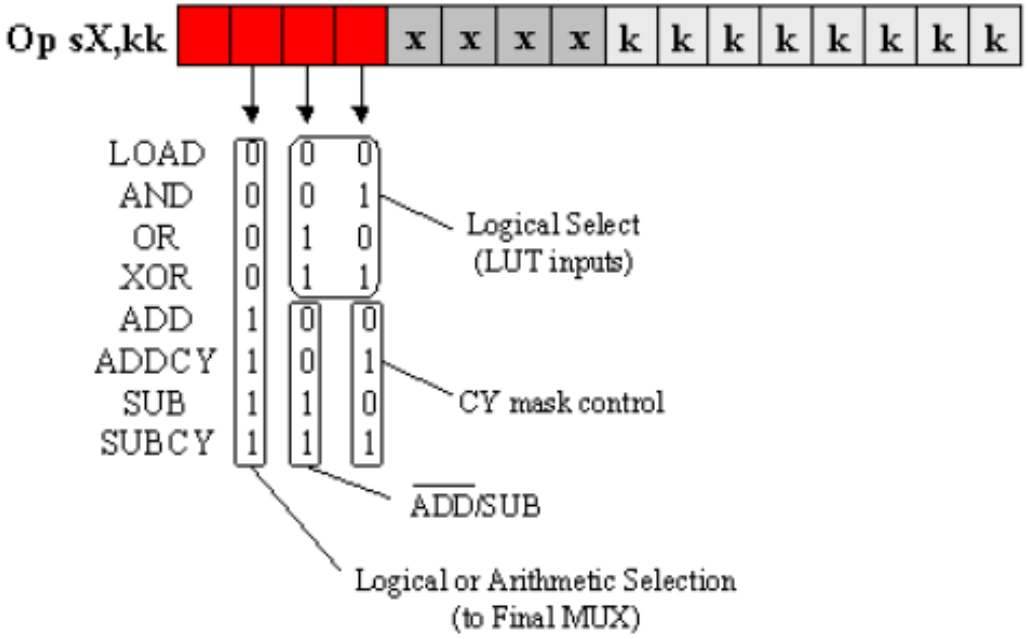


LA ALU


sX
8
Shift & Rotate
5 "Slices"
Logical
4 "Slices"
Arithmetic
4½ "Slices"
MUX
8 "Slices"
8
Result for
sX
sY
kk
8
2nd Operand
MUX
4 "Slices"
2
INPUT
PORT
8

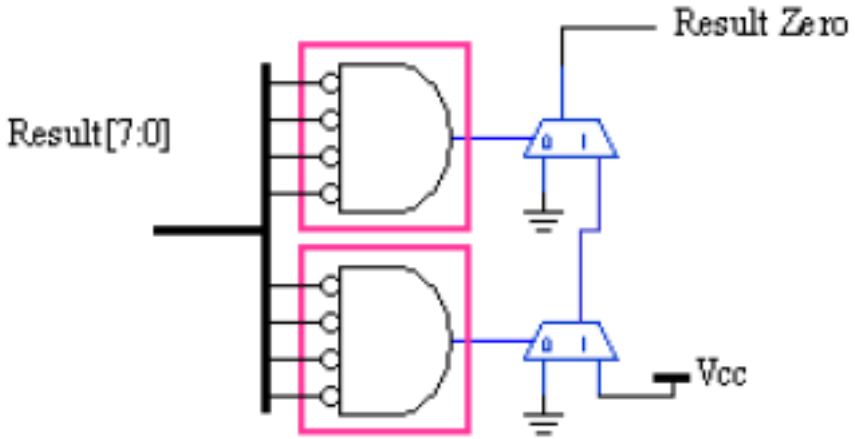
Bits de Selección

- Muchos de los bits del código de operación pueden ser aplicados directamente sobre los bloques de la ALU.



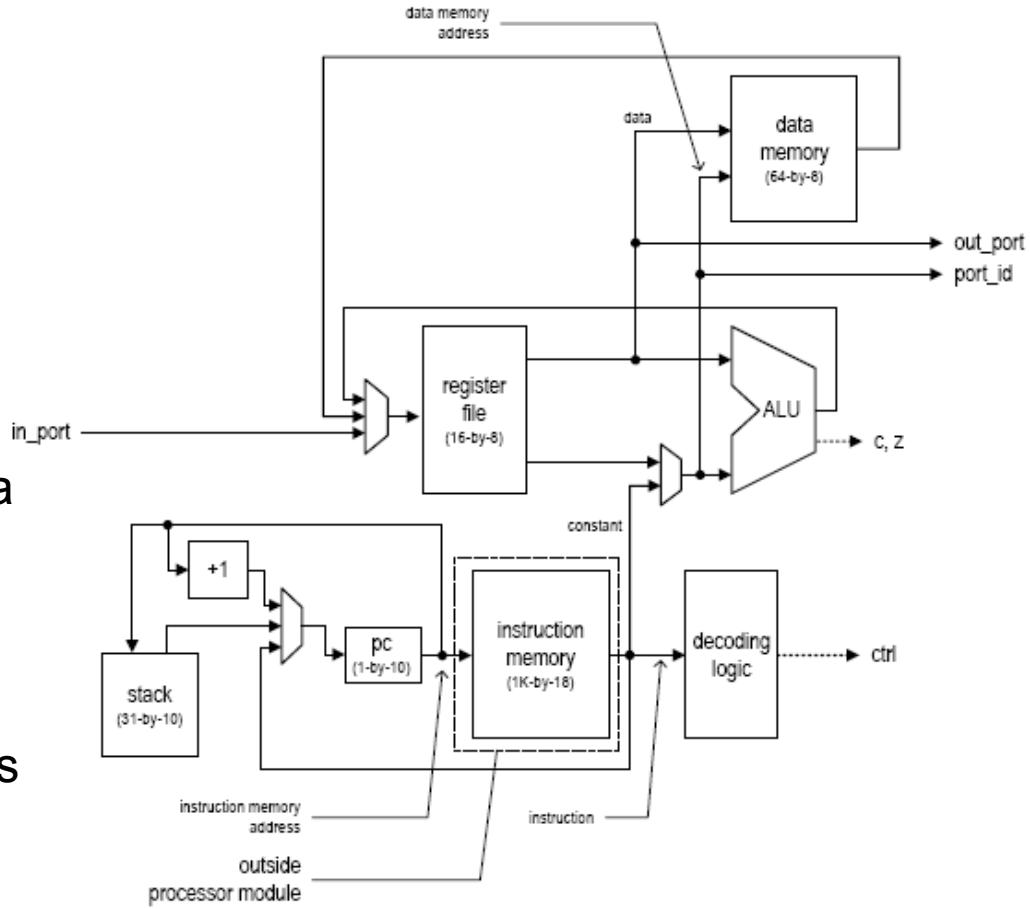
Flags

- Carry: se fuerza a cero en las operaciones lógicas y captura o bien la salida de la cadena de carry en las operaciones aritméticas o bien el bit shifteado durante las operaciones de shift y rotate. O sea, se necesita un multiplexor de 1 bit.
- Zero: se pone a 1 cuando todos los bits del resultado son cero. Es una NOR de 8 entradas.
- Cada flag debe ser implementado usando un clock enable flip-flop para garantizar que solo se actualizan en las operaciones apropiadas.



Organización PicoBlaze

- Datos de 8 bits
- ALU de 8 bits con carry y zero (flags)
- 16 registros generales de 8 bits
- Memoria para Datos: 64 bytes
- Ancho instrucción: 18 bits
- Ancho direcciones de la instrucciones: 10 bits. 1024 palabras para instrucciones.
- Pila de llamada-vuelta subrutina con 32 palabras
- 256 puertos de entrada
- 256 puertos de salida
- CPI instrucción: 2 ciclos
- Atención interrupciones: 5 ciclos



Módulos HDL para PicoBlaze

- Está organizado en dos módulos: KCPSM3 es el procesador (constant (K) coded programmable state machine). El otro módulo es para la memoria de instrucciones. Allí se almacena el código ensamblado.

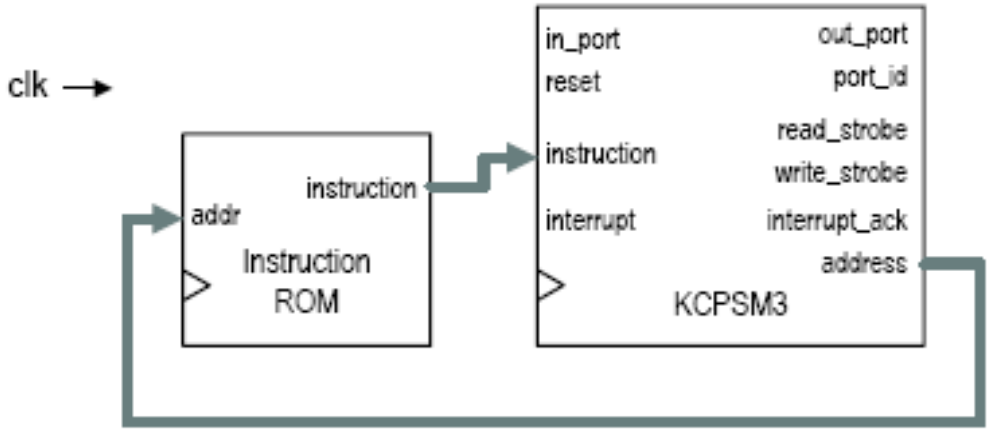


Figure 14.3 Top-level diagram of PicoBlaze.

Señales de PicoBlaze

clk (I)
Reset (I)
address (O): 10 bits
instruction (I): 18 bits
port_id (O): dirección de E/S : 8 bits
in_port (I): data de E/S : 8 bits
read_strobe (O)
out_port (O): dato de 8bits
write_strobe(O)
interrupt(I): solicitud
interrupt_ack (O): ack

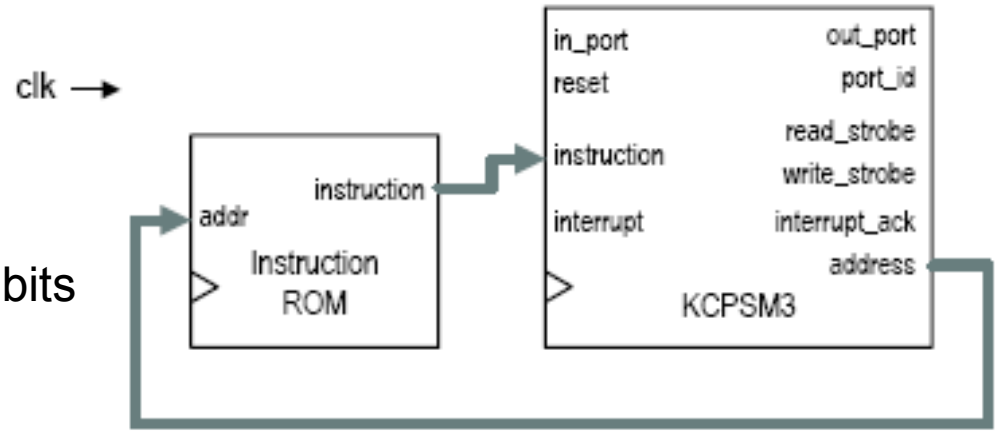


Figure 14.3 Top-level diagram of PicoBlaze.

Desarrollos con Picoblaze

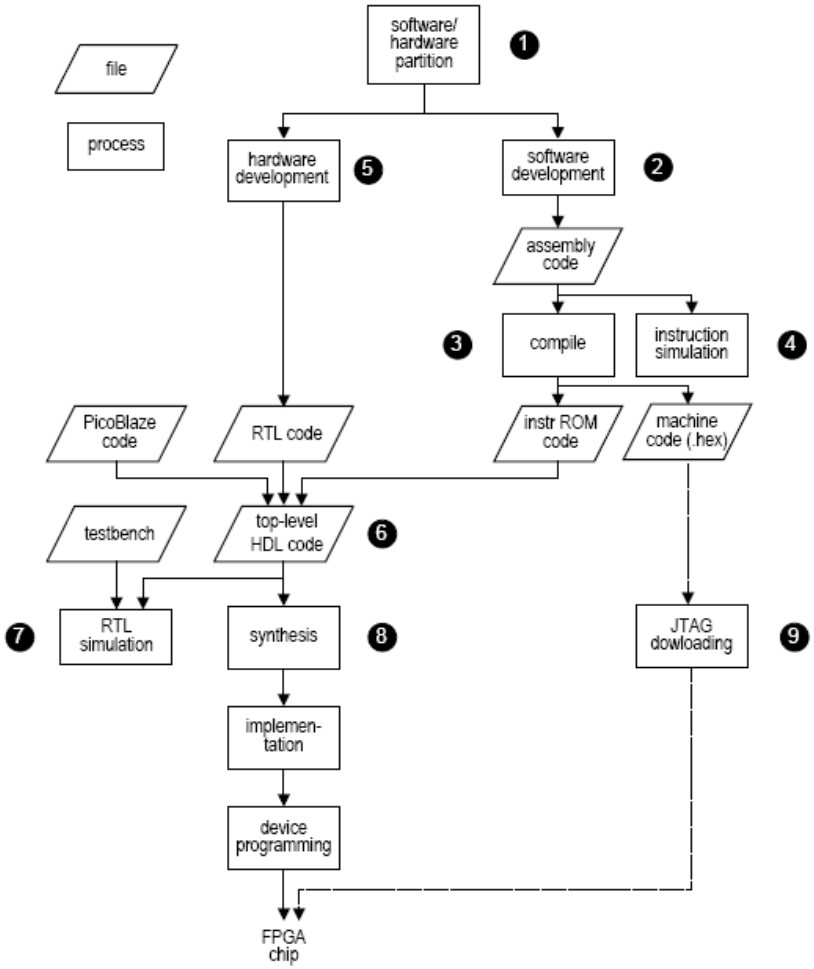


Figure 14.4 Development flow of a system with PicoBlaze.

PicoBlaze: ISA

- 57 instrucciones, cinco formatos.
- Tipos de instrucciones:
  - Lógicas
  - Aritméticas
  - Comparación y Test
  - Shift y Rotate
  - Movimiento de Datos
  - Control de secuencia
  - Interrupciones
- Registros: 16 de 8 bits
- RAM de 64 bytes
- Flags: zero, carry, interrupt
- PC
- TOS (Top of Stack pointer)

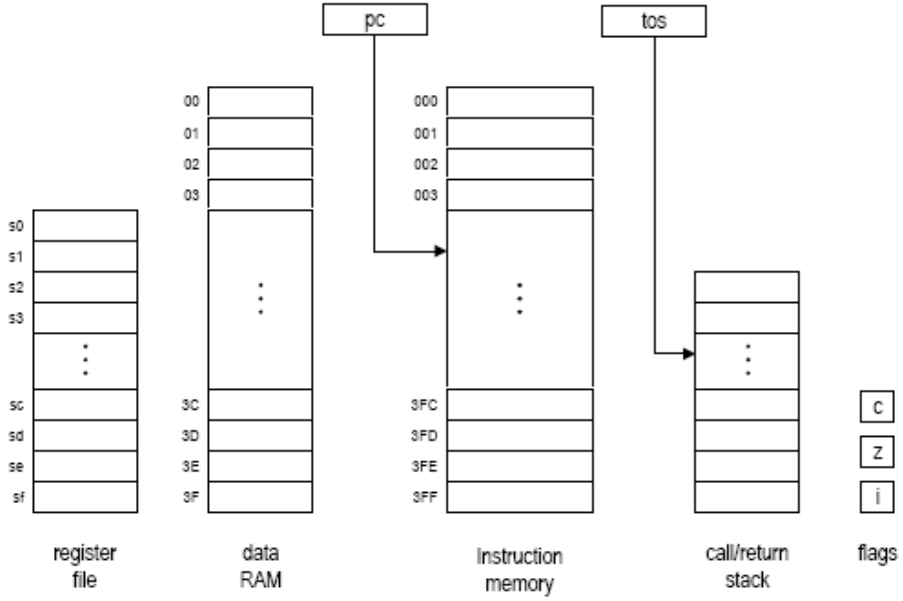


Figure 14.5 PicoBlaze programming model.