Programación del Parallax Propeller usando Lenguaje de Maquina

Una ayuda tutorial de nivel intermedio por deSilva © 2007 Version 1.21 2007-08-21

"Una traducción auténtica de Joe "Bot" Red, autorizada por el autor" por favor enviar comentarios/sugerencias a josezen1@yahoo.com gracias!

Versión 1.21 2007-08-21

Prefacio

Siempre han habido peticiones de ayuda sobre el Lenguaje de Maquina de Propeller. Por supuesto todo está muy bien explicado en la excelente documentación de Parallax; el estilo didáctico de Parallax sin embargo pareciera estar más enfocado en cómo usar SPIN con las características de hardware del Propeller.
Pero el programador avanzado reconoce rápidamente (entre media hora a un par de semanas) que tiene que encontrar su camino en la programación de lenguaje maquina cuando quiere hacer algo más que LEDs parpadeando o uso de "objetos" prefabricados.

Esta ayuda tutorial no fue escrita para el principiante: Usted tiene que tener un buen entendimiento de la arquitectura del Propeller y experiencias exitosas programando en SPIN. Por supuesto Usted debe saber cómo trabajar con el PropellerTool (el Programa de Interfaz del Propeller o IDE) y tal vez el muy útil PropTerminal de Ariba.

Mi intención no es empezar "desde el principio", sino ayudarle en sobrepasar las primeras frustraciones causadas por las peculiaridades del lenguaje de maquina del Propeller
Solo recientemente descubrí que por más de un año Phil Pilgrim ha preparado su "Propeller Tricks and Traps" "Propeller Trucos y Trampas" http://forums.parallax.com/forums/attach.aspx?a=14933 en una manera complementaria a esta ayuda tutorial. Usted se beneficiara enormemente con esta ayuda de Phil después que haya abarcado los primeros tres o cuatro capítulos de este tutorial!, alguno de sus "Trucos" seguramente aparecerán dentro de mi capitulo aun no escrito: "Mejores Practicas"

Como yo programe mi primer micro procesador hace mas de 30 años, y ese no fue mi primer código de maquina con el que tuve contacto, Usted puede que me vea influenciado negativamente a veces. Por Favor perdóneme!.. Yo estoy dispuesto a escuchar sugerencias sobre como improvisar esta ayuda tutorial. Simplemente envié un mensaje al foro o envíeme un mensaje privado!

Y ahora, a divertirnos!

Versiones

1.11 Problemas con esquema wrt resuelto;
    Empecé un apéndice para SPIN
1.20 Problema entendiendo instrucción wrt MUX - resuelto

Como la arquitectura del Propeller difiere considerablemente de otros controladores, yo debería repetir rápidamente sus características y componentes principales. Esto por supuesto ha sido bien explicado en la hoja de datos "Data Sheet" y el manual del Propeller, así que por favor no luzca muy desilusionado!, a través de este tutorial yo debería presentarle un poco más de lo que Usted encontrara en la excelente documentación oficial. Pero yo se la presentare en una forma diferente.
Artículo Anexo A:

Articulo Anexo A: De que esta hecho el Propeller

Existe un 32k ROM, de poco interés para nosotros en los primeros capítulos, además::

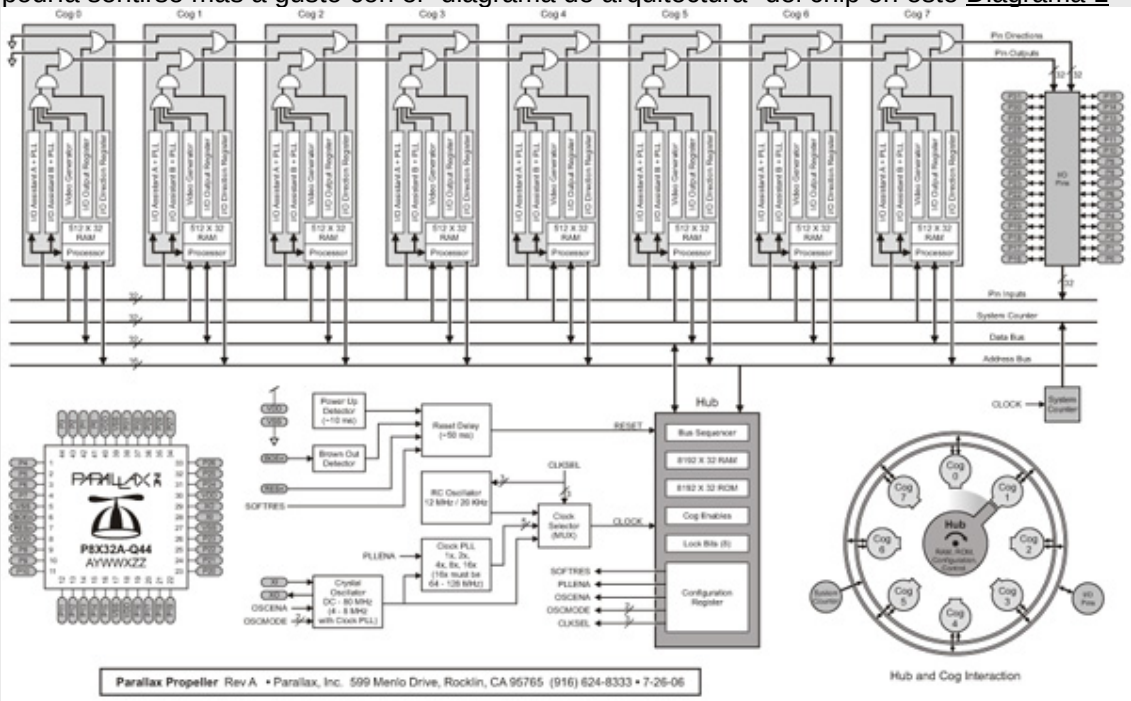
- 32 KB RAM
- -8 procesadores („COGs“) cada uno a 20
- un puerto I/O de 32 Bits (“INA, OUTA, DIRA”)
- un reloj del sistema („CNT“)
- 8 semáforos (“LOCKs”)

Y en cada uno de los 8 COGs:

- 2 KB (512 x 32-bit celdas) RAM estática súper rápida
- 2 temporizadores/contadores (“CFGx”, “PHSx”, “FRQx”, donde x = A o B)
- un procesador de video (“VCFG”, “VSCL”, conectado a Temporizador A)

Esto se resume a:
160 x 32-bit MIPS - 48 kB RAM estática- 16 x 32-bit temporizadores/contadores - - 8-estados lógica de video

Si Usted pertenece al 75% de las personas más orientadas visualmente de este mundo, Usted podría sentirse más a gusto con el "diagrama de arquitectura" del chip en este Diagrama 1



Si no lo ha hecho aun, tome su tiempo para entender TODOS LOS DETALLES!! (Note: el diagrama está conectado a un este web-link hires pdf. O simplemente visite la página Parallax!)

Cuando programe in lenguaje de maquina Usted generalmente debe estar muy claro de todos los conceptos de hardware: la interfaz COG-HUB, temporización exacta, uso de temporizadores/contadores, el "bootstrap" yo incluiré secciones explicando alguno de estos conceptos algunas veces como "Articulo Anexo"

Articulo Anexo B: Que pasa durante RESET/Power On?

❶ Una parte del ROM es copiada en "COG" #0: Este es el Bootstrap Loader, o cargador del Bootstrap. Este se conecta a los pines 30+31 para comunicarse de manera serial con el Propeller IDE (Programa Propeller) o algún otro dispositivo que use el mismo protocolo. (Nota: Este protocolo es abierto y disponible, pero su uso es sin embargo un tanto difícil) Los datos recibidos desde el IDE son entonces almacenados en el HUB-RAM. Opcionalmente los datos pueden ser movidos al EEPROM conectado a los pines 28+29.

Sin embargo, esta conexión puede fallar!
En ese caso:

❷ Los 32 kB de la parte baja del EEPROM serial, conectado a los pines 28+29 son movidos al RAM.

Si esto también falla el Propeller va a modo de mínimo consumo de energía hasta que ocurra un reseteo o nuevo encendido. De otra manera ahora tenemos datos definidos in el HUB-RAM - copiados desde el EEPROM o recibidos a través de la conexión serial - que asumimos sera un PROGRAMA!, Alas, el Propeller no puede ejecutar programas desde el HUB-RAM!

❸ Durante el próximo paso del Boostrap, otra parte del ROM - el Interpretador SPIN (Tamaño 2KB!) es copiado dentro procesador COG #0 y finalmente! - este programa empieza - desde la dirección de memoria 16 en adelante del HUB - a interpreter lo que asume es código SPIN traducido!

Uff!

Hablemos acerca de los procesadores - llamados "COGs" in la terminología Propeller. Que hacen? Siempre hay una repuesta correcta para esto: Ellos ejecutan instrucciones! Un procesador estándar obtiene estas instrucciones desde una memoria accedida globalmente (en la arquitectura llamada "von-Neumann") o desde una memoria dedicada para instrucciones - esta es la manera PICs y AVRs están organizados!). El tener dos memorias permite "entonarlas" de acuerdo a necesidades especificas (por ejemplo: no volátil, solo-lectura, tiempo de acceso rápido) y también accedidas en paralelo!

Un procesador Propeller obtiene sus instrucciones desde la memoria interna del COG, un espacio limitado a 496 instrucciones!, ahora, no corras darle tu Prop a tu sobrino para que juegue! Recuerda, tienes 8 de esos COGs y la memoria COG is RAM, asi que puede ser recargada!, además, tenemos instrucciones de 32 bits, dándole mucho más poder que las instrucciones comunes de 8 bits.

Así que parece que tenemos una arquitectura von-Neumann perfecta, donde las instrucciones y datos son esparcidos en una memoria.

Cada instrucción es de 32 bits y los datos -también son de 32 bits. Ahora, esto es gracioso!, o es que la memoria no consiste de Bytes??

No, no es! Consiste de pequeñas partículas eléctricas almacenadas en estructuras semiconductoras ☺ y ALGUNAS VECES empacadas de ocho en ocho. La memoria COG es empacada in tamaños de 32. Ni más ni menos!

Mejor llamamos estos paquetes "Celdas" para evitar confusiones! Así que tenemos 496 celdas multipropósito, algunas contendrán nuestro programa, algunas contendrán datos, existen 16 celdas adicionales usadas como registros I/O; discutiremos eso mas tarde.

Yo se que estas absolutamente loco por ejecutar tu primera instrucción, pero se paciente! Primero debes aprender como nuestra instrucción va a parar dentro de una celda de uno de los COGs.

Articulo Anexo C: Cargando los COGS

Nosotros dejamos nuestro último Articulo Anexo con el Interpretador SPIN corriendo en COG #0, empezando a leer desde la memoria del HUB. Y tiene que ser código de Byte de SPIN, generado por el Propeller IDE, y ningún otro!, asi que que lo que necesitamos es una instrucción de SPIN que cargara nuestra anunciada instrucción MAQUINA dentro de la "maquina"., quiero decir, en una de las celdas internas de un COG. Afortunadamente ya sabemos algo como eso: se llama COGNEW y empieza una nueva versión del Interpretador SPIN en un nuevo COG, para interpretar una rutina SPIN especifica.

Hey, pero no es eso lo que queremos hacer? Cierto!, pero por razones solo conocidas por los inventores, el cargar nuestro código de maquina en un COG también es llamado COGNEW. El primer parámetro es una dirección del HUB, el segundo parámetro un valor arbitrario que podemos usar a nuestro bene placito.

```
COGNEW(@myCode,0)
```

Esta instrucción de SPIN inicia el copiado de aproximadamente 2000 bytes, empezando en @myCode dentro de las celdas del próximo COG disponible. Esto es básicamente una característica de hardware del Propeller (de otra manera, como podría empezar la rutina de Bootstrap en primer lugar!) No necesita supervisión de ningún tipo. Una de las consecuencias de este comportamiento tan elemental es que siempre cargará un COG completamente, sin entender el significado o uso de esos bits que esta copiando.
Note que esto consecuentemente tomará 500*16/80_000_000 = 100 micro segundos, pero el Interpretador de SPIN continuará su tarea mientras tanto en paralelo, ejecutando hasta 20 instrucciones de SPIN.

También note que ambos parámetros de COGNEW deben ser múltiplos de 4. Ye se que te olvidarás de esto inmediatamente, pero al menos te lo advertí!

Te puedo escuchar desesperado, “PERO Y ACERCA DE MI CODIGO?” Por favor se paciente, vamos a trabajar en eso muy pronto.

El IDE Propeller sabe dos lenguajes diferentes, SPIN y lenguaje maquina (o código maquina). Código de Maquina es encapsulado en las secciones DAT, donde no código SPIN es permitido. Por razones que explicaré después, SIEMPRE empezaremos nuestras secciones de código maquina con:

```
  ORG 0
```

Y las terminaremos con:

```
  FIT 496
```

Ambas instrucciones no son código de máquina. Estas son llamadas directivas de assembler, y existen muy pocas de ellas, de hecho la única otra es **RES.**

En las secciones DAT podemos usar los nombres de todas las constantes definidas del objeto siempre y cuando tengan sentido. Notablemente podemos usar los nombres de todas las características de I/O. o sea, los registros de I/O: **INA, OUTA, DIRA, VCFG, VSCL, PHSA, PHSB, FRQA, FRQB, CFGA, CFGB**.

Asi que, empecemos!

```
PUB ex01
   cognew(@ex01A, 0)

DAT
       ORG  0
ex01A
       MOV  DIRA,   #$FF   '(Cell 0) Salida I/O 0 a 7
       MOV  pattern, #0    '(Cell 1) Inicializar  “registros”
loop
       MOV  OUTA, pattern  '(Cell 2) Salida de patrón a P0..P7
       ADD  pattern, #1    '(Cell 3) Incrementar el „”registro“
       JMP #loop           '(Cell 4) repetir bucle

pattern LONG $AAAAAAAA      '(Cell 5)
       FIT 496
```

Antes de que corras este programa, debes estar seguro que no tienes nada caro conectado a los pines 0 a 7!. El Hydra tiene un LED en Pin 0 el cual encenderá y un conectar de audio en el pin 7, el cual es muy conveniente.

Antes de que veamos los pines usando un osciloscopio o un contador de frecuencia, debemos hacer algunos cálculos, el reloj por defecto RCFAST es 12 MHz, con algunas notables excepciones cada instrucción de maquina toma 4 ciclos de reloj (Recuerda esto!) así que tenemos 333 instrucciones por nanosegundo: MOV, ADD, JMP. Así que el bucle toma **exactamente** 1 microsegundo. Asi que deberíamos observar las siguientes señales:

P0 : 500 kHz
P1 : 250 kHz
...
P7 : 3.9 kHz

Desviaciones cerca del 3% es normal con el reloj RC.

Esto es rápido! e imagine, podemos correr el Prop inclusive 7 veces más rápido!

Ahora, vamos a "investigar" nuestro programa!
Vemos algunas "instrucciones move" llamadas **MOV**; que tienen dos parámetros (u operadores). Llamamos el de lado izquierdo "dest" y el del lado derecho "source". Asi que es obvio que todo se mueve del derecha a izquierda: esto es exactamente como tu escribes tus operadores en SPIN (o como en la mayoría de los otros lenguajes).

Cuando ya tengas experiencia con lenguaje de máquina de algún microprocesador común (8051, 68000, AVR, PIC) esperarás aprender algo acerca "modos de direccionamiento" "registros" etc. De hecho aprenderás!.

Existen dos tendencias de pensamiento: Una (solo yo y la "Hoja de Datos!) dice: existen 512 registros en un COG. La otra tendencia (el resto del mundo!) dice: No existen registros en un COG, excepto los 16 registros mapeados en las direcciones 496 a 511.

Esto no es un problema si tu no sigues my forma de pensamiento, tu puedes fácilmente traducir esto en tu propia versión.

Así que veamos la instrucción **MOV** in la *Celda 2:* copia el contenido del registro 5, en otras palabras el *patrón* dentro del registro $1F4 en otras palabras OUTA. La instrucción **MOV** en la *Celda 0* copia el numero 0 dentro del registro 5. Estos son los dos modos de direccionamiento disponibles en el lenguaje de maquina del Prop: direccionamiento de registros y direccionamiento inmediato. (Pero veras muy pronto que esto es solamente el 97% de la verdad: existen algunas instrucciones que pueden mover data desde y hacia la memoria del HUB!)

Cada una de las instrucciones es capaz de ejecutar este direccionamiento inmediato en el **operador del lado derecho** al colocar el símbolo "#" al frente del operador, aunque esto es lógicamente parte del código de operación.

Que mas tenemos? Ah, también existe una instrucción **ADD**! Es obvio lo que hace: suma 1 al registro 5.
And nothing more obvious than **JMP**, however … Why do we have this funny "#" here, too?? A typo?
Y nada más obvio que la instrucción **JMP**, sin embargo, Por que tenemos este simpático "#" símbolo también? Error?

No - piensa claramente! Cuando usamos el *patrón* en las instrucciones ADD y MOV, queríamos que el procesador VIESE DENTRO de este registró para cargar o almacenar ese valor. Cuando escribimos #1 (en ADD), lo que queremos es que el procesador use precisamente este valor!.
Así que es lo que queremos que el procesador haga cuando saltamos? NO que busque el valor de algún registro, sino que salte a precisamente la celda especificada en el valor: #*loop*.

Pero! También podríamos pedir al procesador que salte a algún destino "computado" que previamente almacenamos en algún registro. Esto es generalmente llamado salto indirecto. Esto es un concepto muy importante, esencial para llamadas a subrutinas.

Es ciertamente fácil para el principiante olvidar el "#", y como de todas maneras es código correcto no será detectado automáticamente. Si tu programa termina de modo nesperado, primero chequea todas tus instrucciones JMPs en busca de este error!.

La última línea del programa es familiar: esta es la manera que utilizamos la sección DAT antes. Para definir y dar valores iníciales a las variables. Pero note que después que la sección DAT ha sido copiada en un COG (a través de la instrucción COGNEW) el procesador ve esta sección de una manera diferente a como es vista por el interpretador SPIN en el "arquetipo" en el HUB!. Para el COG es el "registro 5"; ve tu mismo donde puede estar en el HUB; simplemente presiona F8 y estudia el mapa de memoria. Yo predefiní el valor como $AAAAAAAA para que así puedas encontrarlo más rápido.

Para finalizar este capítulo y antes de continuar explicando mas instrucciones and técnicas de programación deberíamos memorizar la estructura de los 32 bits de una instrucción:

6 Bits: instrucción o código de operacion (OPCODE)
3 Bits: configuración de semáforos (Z, C) y result
1 Bit : direccionamiento inmediato
4 Bits: condición de ejecución
9 Bits: dest - registro
9 Bits: source registro o valor inmediato

Usted no debería realmente memorizar esto!. Esto debería más que todo darle una idea que existe y que no existe dentro de una de estas diminutas instrucciones, de manera que puedas entender algunas limitaciones.

Como ves el rango de un valor inmediato está restringido (desde 0 a 511). Esto no es una limitación para las instrucciones JMP, ya que este rango es exactamente del tamaño del COG. Pero si tu quieres definir o añadir otros valores entonces tienes que pre configurarlos dentro de una celda dedicada, de la misma manera como hicimos en el ejemplo (LONG $AAAAAAAA). Extrañamente esto no toma tiempo adicional! Puedes que estés acostumbrado a que en otros procesadores el método de direccionamiento inmediato sea MUCHO MÁS eficiente que direccionamiento directo. Pero no es así en el Prop, ya que direccionamiento directo es simplemente direccionamiento de registros!!!

Y no te preocupes acerca de lo que aun no entiendas, clarificación vendrá en los próximos capítulos.

Interludio 1: la Sintaxis del Lenguaje Assembly del Propeller

Te has alimentado con tu primer programa de lenguaje maquina ex01 – ya lo digeriste? Puedes que tengas preguntas, si no has visto código como este antes.

La manera como escribes lenguaje de maquina en la forma de una programa de assembler es muy similar a través de todas las computadoras. Pero no igual. Inclusive existe un estándar de cómo escribir código assembly. Aunque poco saben sobre esto y a nadie le importa.

El principio básico es escribir una instrucción por línea, los elementos de esta instrucción son: etiquetas, opcode, operadores, sufijos y prefijos están separados por espacios, tabulación o comas, Una coma es generalmente utilizada cuando el elemento a la izquierda de esta puede contener espacios de forma natural, ejemplo: cuando se escriba una formula constante te gustaría tener esta libertad.

También puedes definir y pre configurar celdas de datos. Generalmente esta pre configuración puede ser "encadenada" separada por comas por la razón expuesta anteriormente. Programadores en SPIN deberían sentirse en casa ya que todo es exacto a como es en SPIN.

La misma situación aplica a los comentarios.

Generalmente existe algo llamado “directivas”, que no conllevan a código o datos pero más bien le indica al assembler que “organice” cosas. Una "directiva" típica podría ser una definición de una constante, pero esto es logrado independientemente en la sección CON.

“Macro-Assemblers” pueden tener hasta cien directivas; pero solo hay tres directivas en el Propeller

```
ORG 0   ‘ re empezar a  “contar celdas” en 0
FIT n   ‘ alarma cuando una cuenta reciente de celdas sobrepasa n
RES n   ‘ incrementar cuenta de celdas por n sin reservar memoria HUB
```

Algunas reglas importantes:

- Use ORG con 0 solamente
- No trate de asignar instrucciones o datos después que uses RES
- Siempre termina con FIT 496

Si tu eres uno de esos tecnócratas empecinados como yo, podrías estar interesado en lo que es llamado “Sintaxis” del lenguaje assembler, Ha existido un refinado sistema por los últimos 50 años llamado BNF (“Backus-Naur Formalism”)

```
directive     ::= ORG 0 | FIT constant | resDirective
resDirective  ::= [label] RES constant
label         ::= localLabel | globalLabel
localLabel    ::= ":"identifier
globalLabel   ::= identifier
number        ::= decimal | hexadecimal | binary | quaterny
constant      ::= constantName | number | constantFormula
constantName  ::= label | nameFromCON

instruction   ::= [ label ]
                  [ prefix ] opcode [ dest "," ] source [postfix]*
prefix        ::= IF_C | …
opcode        ::= MOV | …
dest          ::= constant
source        ::= [ "#" ]constant
postfix       ::= WZ | WC | NR

dataItem      ::= [ label ] size constant [ "["constant"]" ]
                  [ ","constant [ "["constant"]" ] ]*
size          ::= LONG | WORD | BYTE

program       ::=
              [ ORG 0 ]
              [ label | instruction | dataItem ]*
              [ resDirective ]*
              [ FIT constant ]
```

Desde el punto de vista técnico, el Prop posee un set de instrucciones de dos direcciones, conteniendo una sistemática “opción para direccionamiento inmediato” No existen otros modos de direccionamiento sistemático como los que existen en otros procesadores (indexado, pre/pos-in/decremento) Si necesitaríamos este sistema (y de hecho lo necesitamos!) tendríamos que “modificar“ instrucciones, de manera que computamos la dirección requerida y la “implantamos” dentro de una instrucción existente. La ciencia de la computación no le ha dado importancia a esto por décadas. El gran Edsgar Dijkstra esta supuesto de haber escrito un artículo titulado “Código auto modificable considerado dañino”, pero el manuscrito se ha perdido. El Propeller como siempre no puede vivir sin esto; existen inclusive tres útiles instrucciones para soportar este método, llamadas MOVI, MOVS y MOVD. Trabajaremos en ejemplos sobre este tema en el capítulo 5.

Articulo Anexo D: Quien le tiene miedo a OUTA?
Reconsiderando nuestro primer ejemplo **ex01:** si entendemos este concepto de COGs de procesadores paralelos como mostramos en el **diagrama 1** correctamente, hay más cosas ocurriendo dentro del chip que solo el contador de 8 bits en nuestro COG. Cierto, nos aseguramos que podemos jugar con los pines 0 al 7, pero el proceso de incremento también modifica bits más significativos en OUTA….

Las reglas son:
- Un pin físico es **habilitado para salida**, cuando el bit correspondiente en DIRA en **al menos un COG** es puesto a 1.

- Un pin fisico, habilitado como salida, es **puesto a lógico verdadero** cuando el Bit correspondiente en OUTA en **al menos un COG** es puesto a 1.

Bueno… casi...
La segunda parte correctamente debería ser:
- Un pin fisico, habilitado para salida, es puesto a lógico verdadero cuando el bit correspondiente en OUTA en **al menos uno de los COGS donde a sido habilitado para salida** es puesto a 1.

Lo que simplemente significa todo es como esperábamos. Si tienes dudas consulta la versión de alta resolución del Diagrama 1, las compuertas AND y OR relevantes están dibujadas en gran detalle!

Así que ahora podemos comunicarnos con el espacio exterior a través de ondas cuadradas, pero como podemos comunicarnos con el “espacio interior”, el gran área de memoria de 32 kb del HUB? Como podemos nosotros ejecutar instrucciones desde esa área o como tener acceso a los datos?

Pusiste atención? Los procesadores COG extraen sus instrucciones solo desde su memoria COG. Sin excepciones! Eso significa, SI nosotros queremos tener programas más grandes que lo que cabe en la memoria COG tendrías que recargarlos. Esto es complicado y el cómo hacerlo eficientemente será parte de la ayuda tutorial “Nivel Maestro” ☺ puedes encontrar algunas reseñas un tanto cripticas en el foro de Parallax, has búsqueda por "LMM: Large Memory Model".

Pero leer o escribir desde y hacia el HUB es muy fácil. Existe un set de 6 instrucciones específicas para esto (Por cierto: esto ha sido etiquetado como una "Arquitectura de Carga-Almacenamiento" en el jargón de Ciencias de Computación):

- WRBYTE y RDBYTE
- WRWORD y RDWORD
- WRLONG y RDLONG

Para ser usado de manera muy directa:

```
RDBYTE Celda_en_COG, dirección_en_HUB
```

Pero como sabemos que direcciones de HUB son apropiadas? Existen dos posibilidades:

- 1) Usted puede proveer un parámetro a COGNEW, el cual convencionalmente es un apuntador a alguna posición de la memoria HUB, perfecta para ser usada por, digamos, RDBYTE. Este parámetro is "automáticamente" copiado en la celda 496 del COG y puede ser referenciado simbólicamente por el nombre **PAR** (Recuerde: tiene que ser un múltiplo de 4!)

- 2) La segunda opción es más difícil. Recuerde que el código a ser cargado en el COG es siempre parte de la memoria HUB primero (llamemos esto el "arquetipo", como ya dijimos anteriormente). Así que puede ser modificado por instrucciones SPIN (Imagine: es muy simple escribir Assembler en SPIN). Pero en cualquier caso podemos asignar algunas de las variables DAT **antes** que sean cargadas en el COG

Confundido? Aquí hay un ejemplo #2 para clarificar las ideas:

```
VAR
  LONG aCounter

PUB ex02
  patternaddr := @aCounter
  COGNEW(@ex02A, 0)
  COGNEW(@ex02B, @aCounter)
  REPEAT
    aCounter++

DAT
      ORG    0
ex02A
      MOV    DIRA, #$FF  ' Pines 0 al 7  como salidas
:loop
      RDLONG OUTA, patternAddr
      JMP    #:loop
patternAddr
      LONG   0         ' dirección de la variable usada para comunicación
                       ' debe ser almacenada aquí antes de cargar el COG

      ORG    0
ex02B
      MOV    DIRA, altPins ' Pines 8  al  15   para salida de resultados
```

```
:loop
        RDLONG r1, PAR
        SHL  r1, #8
        MOV  OUTA, r1
        JMP    #:loop

altPins LONG $FF<<8
r1      LONG 0
```

Yo pensé que este momento es tan bueno como cualquier otro momento para introducir nuevos conceptos. Los métodos (1) y (2) usados para establecer comunicación deben estar claros como la luz del día en este momento. Por supuesto, el segundo COG que activamos tiene que evitar el uso de los pines 0 al 7. Así que movimos el área de actividad desde el pin 8 hacia arriba. Estos pequeños cambios pueden tener consecuencias imprevistas en código de maquina: ahora no es posible copiar el patrón de salida directamente en OUTA, tenemos que "empujarlo" y conoceremos una nueva instrucción para esto, de hecho, existe una familia completa de instrucciones similares, las cuales consisten en:

```
SHL : empujar hacia la izquierda, rellenando con ceros
SHR : empujar hacia la derecha, rellenando con ceros
SAR : empujar hacia la derecha, rellenando  bit 31
ROR : empujar hacia la derecha (i.e. bit 0 conectando al  bit 31)
ROL : empujar hacia la derecha (i.e. bit 0 conectando al  bit 31)
RCL : rotar con acarreo hacia la izquierda
RCR : rotar con acarreo hacia la derecha
```

También debemos introducir una nueva celda intermediaria “r1” – este es el uso típico de un “registro”, que usamos entre dos o tres instrucciones y después lo desechamos. Es mejor utilizar una “convención de nombres” para este tipo de celdas, especialmente cuando tus programas se hacen más grandes. “r1”...”r99” o “A”..”Z” – debes tratar de ser consistente a través de todos tus programas.

Notaste el “.”? De seguro, pero que significa? He hecho “no tiene significado”. Simplemente ayuda a olvidarlo después de su uso, así que este nombre puede ser re usado en otro contexto. Esto es muy ventajoso para las personas menos imaginativas que tienden a llamar sus etiquetas “lab”, “bucle” “rep” así por el estilo. El rango de estos nombres “locales” es desde una etiqueta “global” a la próxima etiqueta “global”

Algo que no notaras inmediatamente es, que RDLONG toma mucho más que 4 ciclos de reloj .

Articulo Anexo E: Porque el HUB es llamado el HUB

Aunque suene trivial: Casi todos los aspectos del Propeller son mostrados en el diagrama 1 “diagrama de arquitectura”. Allí ves el HUB y los 8 COGs: el HUB rota y se conecta a un HUB cada 2 ciclos de reloj, sumando hasta un total de 16 ciclos hasta que retorna al mismo COG.

La ciencia del procesamiento paralelo – de más de 30 años – ha ideado muchos esquemas de comunicación entre dispositivos paralelos: buses, entrelazados… El mecanismo del COG de ranuras de tiempo sincronizadas es el más básico (y estable!). Se asemeja al “protocolo de bus token” en teoría general de comunicaciones. Y es un desperdicio del ancho de banda.

Pero no estamos aquí para criticar, sino para entender. Así que cada 16 ciclos de reloj un COG es capaz de leer o escribir a la “memoria principal”. Cada vez que trata de hacer esto “des Sincronizado” tiene que esperar. Esta es la razón por la cual la sincronización de las instrucciones de HUB RD…, WR… y otras es bastante imprecisa: les toma entre 7 y 22 ciclos de reloj, dependiendo en donde la rueda del COG se encuentre al momento que esta instrucción es ejecutada dentro del COG.

Una vez que hayas realizado una transferencia a memoria satisfactoriamente, estarás “sincronizado”, ósea, ya sabes que te podrás conectar otra vez exactamente 16 ciclos después. Esto te deja con solo dos instrucciones intermediarias (=8 ciclos mas 7+ ciclos) para leer otra vez o escribir a la memoria del HUB

O Bien, saca tu osciloscopio otra vez o conecta altavoces al pin 7!.. pero que paso cuando nuestra perfecta señal de 500 kHz? Ohh mis estimados. Tu entiendes el por qué?

Por supuesto podemos también escribir de vuelta en la memoria del HUB, la instrucción es WRLONG (WRWORD, WRBYTE respectivamente)

```
  WRLONG cogCell, hubAddress
```

Note que los operadores están en el mismo orden que RDLONG. Lo que significa que el flujo de datos es ahora de izquierda a derecha! Esta es la única excepción en el sistema y por esa razón es una fuente común de confusión.

Cuando tengas dudas memoriza lo siquiente: is posible escribir desde y hacia los primeros 512 bytes en la memoria HUB utilizando “direccionamiento inmediato”! aunque esto es raramente utilizado, es parte del “sistema” general. Y valores “inmediatos” solo son permitidos en el operador del lado derecho…

Ahora, regresemos a cosas menos extrañas! Subrutinas son los elementos primordiales de programas complejos y casi tan importantes como las instrucciones JMP. Por supuesto son un tipo "de salto", pero permiten "retornar control al origen".

Si eres un programador assembler experimentado puedo ver chispas en tus ojos: parámetros PUSH a la pila (stack), rutinas recursivas CALL. Y también POP de toda la basura innecesaria!

Oh mi amigo - lo siento! Nada de eso - en serio nada de nada!

No, el Propeller es a real maquina RISC: un ciclo de reloj por instrucción (bueno, cuatro para ser honesto - pero eso pronto cambiara con el Prop II)! Y esto es cierto inclusive para instrucciones CALL. Sabe Usted cuantos ciclos de reloj necesita un AVR mega8 para ejecutar una instrucción CALL? Averigua!

Pero vale la pena pensar por un momento: cómo puedes idear una llamada a subrutina sin una pila (stack)?

Aquí está la respuesta:

```
' ex03A
DAT
      ORG    0
ex03A
      MOV    m10par, #30          ' este numero 30 ...
      JMPRET times10_ret, #times10 ' ... va a ser multiplicado por 10
```

```
' aquí va mas código del programa principal
      .....

' aquí empieza la subrutina
times10
      MOV    m10par2, m10par   ' haz una copia
      SHL    m10par2, #2       ' aquí multiplica x 4
      ADD    m10par, m10par2   ' mas 1 = 5
      ADD    m10par, m10par    ' multiplicado por  2 = 10
      JMP    times10_ret       ' salto indirecto

times10_ret
      LONG   0

m10par  LONG   0
m10par2 LONG   0
```

Esta rutina ejecuta una multiplicación optimizada por 10, de manera bastante directa. Note que necesitamos muchos registros intermediarios (m10par, m10par2) ya que no podemos hacer PUSH o POP.

De nuevo nos encontramos con una nueva instrucción

```
  JMPRET ret, subr
```

Este almacena la dirección de retorno en los 9 bits menos significativos de la celda *ret* y salta al área *subr* – ahora, si este es una etiqueta (generalmente lo es), entonces no olvides el sufijo “#”!

El retorno es muy simple, solo has:

```
  JMP ret
```

Nota que este es un salto indirecto, sin ningún sufijo "#"!

Existen rutinas con múltiples salidas, pero la mayoría solo tienen una. En esta situación podemos utilizar un truco ingenioso, en vez de salir con:

```
          JMP times10_ret
times10_ret  LONG 0
```

Simplemente codificamos:

```
times10_ret  JMP# 0
```

Uff!

Ok, vamos a calmarnos un poco !
Como empezó todo en primer lugar?

```
JMPRET times10_ret , #times10
```

Ok, el código salta a *times10* DESPUES que almaceno la dirección de retorno en la celda *times10_ret*… más precisamente en los **9 BITS MENOS SIGNIFICATIVOS** de la celda *times10_ret*.

Como veras: Esta es la magia del código modificable! Y también puedes entender que necesitamos el sufijo "#" aquí porque ahora no queremos direccionamiento indirecto, ya que la dirección de retorno ya ha sido almacenada en la instrucción.

Y si piensas que esto es terriblemente complicado, tal vez tengas razón.

Deberíamos tomar un descanso ahora, pero antes que pases la noche sin dormir, aquí tienes medicina. Existe un atajo para:
JMPRET subr_ret, #subr – lo cual es lo mismo que: **CALL #subr**

Y – hurra! – también existe un atajo para:
JMP# 0 – el cual es: **RET**

Y si piensas que esto no es medicina sino un placebo, puedes que tengas razón otra vez ☺

Lo que necesitamos ahora es una lista de instrucciones de manera que podamos programar cosas útiles (FFT, gráficos 3D, reconocimiento de voz, decodificación de mp3)

Después que los procesadores empezaron a usar más de 8 instrucciones algunos años atrás. Existe solo una respuesta a este requerimiento RTFM. Lo que podemos hacer en esta ayuda tutorial es mostrar los más importantes y aclarar algunas cosas no obvias.

El elemento más útil es discutiblemente la instrucción “decremento y salto si no es cero” **(DJNZ)**. La mayoría de las cosas funcionan de una manera más o menos diferente en el Propeller comparado con otros micros controladores, pero DJNZ es una notable excepción: funciona EXACTAMENTE como en otros procesadores que tiene esta instrucción.

- **DNJZ** es un “salto condicional”, usado para contar bucles.
“decisiones tipo IF” son puestas en ejecución por otros “saltos condicionales”
- **TJZ** “Prueba y salta si es cero”
- **TJNZ** “Prueba y salta si no es cero”

[i]TBD: Veremos un ejemplo dentro de poco.[/I]

Estas tres instrucciones chequean un registro para determinar si esta o no “vacio”. Todas las otras instrucciones condicionales dependen en las llamadas banderas, las cuales tienes que ser evaluadas por alguna instrucción previa.

Intrucciones condicionales?" Esto es algo nuevo para muchos programadores experimentados! Existen pocos procesadores que ofrecen una instrucción de “sobre salto”. Esto puede ser considerado como un “prefijo de instrucción”, determinando si la instrucción en cuestión debe o no ser ejecutada. Sin embargo este tipo de técnica es generalmente no explicada en clases ya que las instrucciones (compuestas) son hacen bastante largas.

El propeller sin embargo ha incluido el concepto de “ejecución condicional” dentro del sistema. 4 bits de cada instrucción han sido dedicados para este propósito!

Así que si entendemos correctamente. Deberíamos haber mencionado que existen de hecho dos banderas en cada procesador, llamadas **C** (“acarreo”) y **Z** (“Cero”). Estos nombres tienen raíces históricas, ya que desbordamiento aritmético © y “vacio” (Z) son dos aplicaciones importantes de estas banderas. La mayoría de los procesadores tienen más banderas (muchas más), pero el Propeller solo tiene 2. Lo que significa que existen 16 estados (= $2^{(2*2)}$) que pueden ser considerados como una “condición”.

```
Sin Acarreo
Sin Acarreo y  No Cero
Sin Acarreo y Cero
Sin Acarreo o Cero
Sin Acarreo  o  No Cero
Acarreo
Acarreo  y  No Cero
Acarreo  y  Cero
Acarreo  o  Cero
Acarreo  o  No Cero
No Cero
```

```
Cero
Acarreo == Cero
Acarero <> Cero
Nunca
Siempre
```

Inclusive existen nemónicos, de manera que las banderas de ciertas instrucciones como “comparar” (CMP) son seteadas para reflejar las relaciones numéricas
< , > , => , =< , == , <>

Las cuales son combinaciones directas de las “banderas básicas”, pero contienen nemónicos adicionales. El lector deberá consultar la table 5-2 (pag.369) del Manual de Propeller para más detalles.

Repitiendo: todas y cada unas de las instrucciones pueden ser ejecutadas dependiendo de cualquier combinación de las dos Banderas C y Z, esto no toma tiempo de ejecución o memoria adicional. También puedes ejecutar DJNZ o TJZ como una instrucción condicional. Aunque nunca he visto un programa haciendo esto, podría ser útil para algunos programas MUY complejos ☺

Antes de que podamos hacer algunos ejemplos instructivos debemos entender como estas dos banderas con seteadas, reseteadas o dejadas sin modificación. Todos los programadores experimentados en Assembler saben que esto puede ser una pesadilla! Una pequeña instrucción insertada por alguna razón en una cadena de instrucciones puede destruir un algoritmo inteligente y eficiente construido con una bandera. Las instrucciones del 8080 son muy inteligentes, ya que las instrucciones de 8 bits generalmente influencian banderas y no así las instrucciones de 16 bits - con algunas excepciones.

Hay buenas noticias! Te puedes olvidar de estos problemas del pasado, ya que las instrucciones del Propeller solo influencian una bandera, solo si le dices que lo haga! Esto está indicado por notación de “sufijos”: escribes **WC** (“con Acarreo de Bandera”) o WZ (“con bandera cero”) al final de cada instrucción.

Ahora solo tienes que memorizar en que situaciones una instrucción PUEDE setear una bandera (si le es permitido hacerlo) Algunas reglas básicas son:
- Instrucciones para mover, aritméticas y lógicas cambian **Z** independientemente si el resultado es o no es cero.
- Instrucciones aritméticas cambian **C** de acuerdo con el “desbordamiento”
- Instrucciones lógicas setean C para formar paridad “uniforme” de todos los bits.

Para el resto de las instrucciones esto es más complejo ☺

Después de toda esta teoría necesitamos un ejemplo: Queremos contar todos los bits en estado verdadero en una palabra (imagina es el resultado de empujar la entrada de 32 bits de una señal y estimar el porcentaje de ciclos verdaderos)

```
'ex04A
theWord    long $XXXXXXXXXX
counter    long 0
result     long 0

      MOV result, #0       ' acumulará  el numero de bits
      MOV counter,  #32    ' chequeamos esta cantidad de bits
:loop ROL theWord, #1 WC   ' el Acarreo refleja Bit 31, y rota a la izquierda
      IF_C ADD resultado, #1
      DJNZ counter, #:loop
```

Este programa tiene muchos beneficios: la variable *theWord* no cambia de valor después que el algoritmo ha finalizado; el programa toma un tiempo definido y constante de ejecución, y la acción (ADD#1) puede fácilmente ser intercambiada con otra acción.

Aquí hay una alternativa:

```
'ex04B
theWord    long $XXXXXXXXXX
result     long 0

      MOV result, #0        'acumulará  el numero de bits

:loop
      SHR theWord, #1 WC WZ  'el Acarreo refleja Bit 31, y empuja a la derecha
                             'Z indica si el registro esta vacio
      ADDX result, #0
      IF_NZ JMP #:loop
```

Este programa destruye *theWord*, pero generalmente trabaja más rápido, también podrías eliminar el contador (6 celdas comparado contra 8 celdas en ex04A)

Note que realmente no existe una diferencia entre:

```
   ADDX result, #0
```

y

```
   IF_C ADD result, #1
```

Tu conoces “Perl”? Cierto! “Hay más de una manera de hacerlo” ☺

Cuando aprendemos acerca de un nuevo procesador, una pregunta común is “que puede este procesador hacer que mi procesador viejo no puede hacer?”

(a) “rápido cálculo numérico” no cuentes con esto. De hecho existe un simulador de punto flotante que no es nada malo, pero su proceso es por debajo de los 100 kFLOPS dejando una distancia grande comparado con coprocesadores matemáticos, eso sin mencionar las unidades SIMD ofrecidas en los procesadores de PC empezando con el P3.

(b) Tampoco tenemos instrucciones de multiplicación y división de punto fijo, procesamiento de señales ambiciosos no es posible, aunque algunas aplicaciones de audio son posibles.

No llores! Existe un maravilloso set de instrucciones de 32 bits para computaciones menos ambiciosas pero sin embargo de alto rendimiento:

(c) Existe adición de 32 bit (ADD), substracción (SUB) y comparar (CMP) con signo (…S) al igual que sin signo, inclusive soportando aritmética de precisión múltiple (…X)

(d) **MOV, MOVI, MOVS, MOVD, NEG, ABS** y **ABSNEG**; recuerda que tenemos instrucciones de 2-direcciones por lo que NEG y ABS también pueden hacer copias en otro registro, por ejemplo:

```
NEG regA, #1    ' Aquí simplemente seteamos regA como –1;  muy util!
```

(e) Operaciones lógicas o a nivel de bit (bitwise) (**AND, ANDN, OR, XOR, TEST**)

(f) Un set completo de instrucciones de empuje (shift) de bits, con un valor **arbitrario** (0…31), obtenidos directamente ("inmediato") o a través de un registro - esto es realmente una característica de gran uso! ! (**RCL RCR ROL ROR SHR SHL SAR**)

(g) Ya hemos discutido las instrucciones de salto en el capitulo 3:

TJZ r, jumpdest
TJNZ r, jumpdest

Salta,si "r" esta vacío, o no vacío respectivamente ; note que las "banderas" son usadas o cambiadas.

DJNZ r, dest

Similar a TNJZ, pero la instrucción hace decremento del valor "r" antes de probar el valor. Estas tres instrucciones solo toman 4 ciclos de reloj solo SI saltan, ahora, si continua con la próxima instrucción, entonces la tubería de instrucciones (pipeline) tiene que empezar de nuevo lo cual necesita 4 ciclos de reloj adicionales para estar en fase de nuevo.

(h) Pero el Propeller también ofrece un set de instrucciones raramente encontradas en otros procesadores. Cuando escribas tus primeros programas de maquina es preferible que las evites, ya que el no entender bien estas instrucciones podrían engañarte y llevarte a errores difícil de identificar. Sin embargo existen buenas razones para tener instrucciones, ya que aceleran ciertos tipos de algoritmos considerablemente!

MAX a, clipval

El problema principal con esta instrucción es que ES la operación de MINIMO matemático. La mejor descripción de lo que hace es: "recortar a un valor máximo"; así que recorta "a" a "clipval" si es necesario. Esta es una operación bastante usada en todo tipo de programación de gráficos.

MIN a, clipval
Este es el "recortar a un valor mínimo", promoviendo "a" a "clipval", si su valor fue menor (o sea, es una operación de MAXIMO matemático). Así que se cuidadoso con los nombres que le han dado a estas operaciones!

Ambas instrucciones también están disponibles para valores con signo **(MINS, MAXS)**

CMPSUB a,b
Substrae "b" de "a", pero si y solo si este debe dejar un valor no negativo en "a". Esta instrucción soporta la operación de división; aquí mostramos la aplicación más sencilla:

```
' ex07A
' calcular  c := a dividido por b; c y b  son asumidos como valores positivos
    MOV     c, #0
:loop
    CMPSUB  a,  b WC WZ 'Acarreo es seteado, si la operación es ejecutada
    IF_C        ADD  c, #1
    IF_C_AND_NZ   JMP  #:loop
' el remanente de la división esta en "a"  ahora
```

Tambien podriamos haber codificado (aunque sin ninguna otra ventaja):

```
' ex07B
' calcular  c := a dividido por b; c y b  son asumidos como valores positivos
     NEG     c, #1
:loop
     ADD  c,  #1
     CMPSUB  a,  b WC WZ   'Acarreo es seteado, si la operación es ejecutada
     IF_C_AND_NZ   JMP  #:loop
' el remanente de la división esta en "a" ahora
```

Veremos una versión avanzada de la operación de división después! Puedes imaginar en que se diferencia? Pista: Piensa en como lo aprendiste en la escuela! ☺

(i) Esta es una instrucción muy peculiar:
REV a, n
Inicializa los "n" bits mas significativos e invertir la secuencia de los (32-n) bits menos significativos del registro "a" Invertir? Esto es: bit0 <-> bit 32-n, bit1 <-> bit 32-n-1, etc.

(j) También existe una muy especial instrucción, un tanto aislada:
SUBABS a,b lo que hace es: a := a - |b|

(k) las próximas instrucciones vienen en grupos de 4, ya que sus resultados dependen el seteo dado a una de las banderas, ya sea C, NC, Z o NZ

Cuatro instrucciones de "multiplexado":
MUX* r, *mascara*
SEtea todos los bits en el registro "r" con un correspondiente UNO en la *mascara* con UNO o CERO, dependiendo de la tabla a continuación. Lo siento, he tratado de explicar esto de otras maneras, pero o no funciono o contenia errores☺, Los bits en "r" donde un bit correspondiente en la *mascara* es CERO no son afectados de manera alguna.

```
  |C |NC|Z |NZ  <- *
--+------------
C | 1  0  -  -
NC| 0  1  -  -
Z | -  -  1  0
NZ| -  -  0  1
 ^
 |
Bandera
```

También tenemos 4 instrucciones de "negado" condicionales
NEG* destino, fuente
Dependiendo de las banderas C, NC, Z o NZ puede que un **MOV** o un **NEG** sea ejecutado. El codigo equivalente seria:

```
IF_*  NEG destino, fuente
IF_N* MOV destino, fuente
```

Por último tenemos 4 instrucciones de "balanceo de cuentas"
SUM* sumando, fuente
Dependiendo de las banderas C, NC, Z o NZ puede que un **ADDS** o un **SUBS** sea ejecutado

(i) El último grupo de instrucciones son bien conocidas en SPIN:
CLKSET
COGID COGINIT COGSTOP
LOCKNEW LOCKRET LOCKCLR LOCKSET
WAITCNT

Estas trabajan de manera similar a las instrucciones en SPIN, asi que no existe necesidad de elaborar (tu conoces SPIN, cierto?)

Interludio 2: Algunos ejemplos aritméticos

Yo creo que llego la hora de presentarles código profesional que debería ser posible para Uds. entender ahora, copiados de las librerias de Parallax: multiplicación, división y raíces cuadradas. Yo deje los comentarios originales:

```
' Multiplicar x[15..0] por y[15..0]     (y[31..16] deben ser 0)
' a la salida, el  producto  queda en  y[31..0]
'
mult  shl   x,#16     'poner multiplicando dentro x[31..16]
      mov   t,#16     ' alistar para los 16 bits del multiplicador
      shr   y,#1 wc   ' poner el bit  inicial del multiplicador en c
:loop
      if_c add y,x wc     'conditionalmente añadir multiplicando dentro del producto
      rcr   y,#1 wc         ' poner el próximo bit del multiplicador en c
                            ' mientras los bits del producto son empujados (shift)
      djnz  t,#:loop       ' repetir el bucle hasta finalizar

mult_ret
      ret             'retorna con el producto en y[31..0]
```

Esto es mas corto de lo que pensabas cierto? Solo 7 instrucciones! Pero consume tiempo! Y no es una multiplicación 32x32 sino 16x16. Esto es muy común; la multiplicación en hardware del Prop II seguramente será también 16x16 solamente. Pero fácilmente puedes hacer 32x32 basada en esta. Como? Recuerdas "binomios" en la escuela? No?

Es: $(a+b)^2 = a^2 + 2*a*b + b^2$
El resto es simplemente codificación:

Este algoritmo debería estar claro para ti: es una multiplicación "estándar" de la misma manera que la haces en el sistema decimal. Sin embargo, tiene una gran ventaja: la tabla de multiplicación es solo 1x1=1 ☺

Cuando no entiendas un programa debes "analizarlo" paso a paso: has una lista con columnas con todas las variables relevantes, y escribe - línea por línea - como cambian!

```
' Dividir x[31..0] por y[15..0]          (y[16] debe ser 0)
' a la salida, el cociente esta en x[15..0] y el  remanente  esta en x[31..16]
'
divide   shl y,#15     ' poner el divisor dentro de  y[30..15]
         mov t,#16     ' alistar para los 16 bits del cociente
:loop    cmpsub x,y wc     'si y =< x entonces substraelo, y setea C
         rcl x,#1      'rotar c dentro del cociente, y empujar (shift) el dividendo
         djnz t,#:loop  'repetir el bucle hasta finalizar
divide_ret
         ret           'cociente queda en x[15..0], remanente en . in x[31..16]
```

Esto también debería ser más corto de lo esperado! (6 instrucciones) . Nota el uso inteligente de la instrucción CMPSUB! Como estamos lidiando con el sistema binario, no existe necesidad de poner CMPSUB en un “bucle” (loop) como hicimos en el ejemplo preliminar ex07!. Este algoritmo es estilo “división de escuela” – en el sistema binario.

```
' Calcular raíz cuadrada de y[31..0] dentro de  x[15..0]
'
root          mov     a, #0        'inicializar acumulador
              mov     x, #0        'resetear root
              mov     t, #16       ' alistar para los  16 bits de root

:loop         shl     y, #1   wc   'rotar los dos bits más significativos de  y …
              rcl     a, #1        ' … dentro del acumulador
              shl     y, #1   wc
              rcl     a, #1
              shl     x, #2        'determinar el próximo bit de root
              or      x, #1
              cmpsub  a, x    wc
              shr     x, #2
              rcl     x, #1
              djnz    t, #:loop     'repetir el bucle hasta finalizar

root_ret      ret                  'el resultado de raíz cuadrada queda en  x[15..0]
```

Aquí te lo dejo a tu propio ingenio ☺

Direccionamiento indexado es necesario cuando queremos extraer algún valor de un vector X[I] (direccionamiento indirecto es un caso especial donde I == 0) Otros procesadores utilizan conceptos diferentes para lograr esto, algunas veces limitando el "índice" o la "dirección base" a ocho o dieciséis bits. Pos o Pre incremento del índice también es una opción común. Cerrando esta discusión, no existe nada como esto en el Propeller ☺

Por supuesto hemos aprendido como accesar la memoria HUB usando RDLONG o WRLONG con una dirección pre calculada in un registro del COG. Lo cual podría lucir como :

```
'ex07A
  MOV r, I
  SHL r, #2          ' x 4 = byte de dirección en el  HUB
  ADD r, X
  RDLONG r, r
```

Para familiarizarnos con este tipo de acceso a la memoria vamos a calcular la suma de 20 números ubicados en la memoria del HUB:

```
'ex07B
  MOV addr, X
  MOV sum, #0
  MOV count, #20
:loop
  RDLONG r, addr
  ADDS sum, r
  ADD addr, #4     ' Proximo Long!
  DJNZ count, #:loop
```

Pero como administramos esto cuando el vector de 20 números está ubicado dentro el COG?
Presentado el código auto modificable (self modifying code)!

```
'ex07C
' asumimos que de  X  a  X+19 contienen los 20 longs que vamos a añadir
    MOVS :access, #X  ' esta instrucción modifica una celda del COG (*)
    MOV sum, #0
    MOV count, #20
:loop
:access
    ADDS sum, 0-0     ' los 9 bits menos significativos de este instrucción …
                      ' … serán modificados por (*)
    ADD :access, #1   ' modifica una celda para que apunte al próximo numero
    DJNZ count, #:loop
    …
X:   RES 20
```

El lector alerta habrá notado una nueva instrucción: **MOVS** ¡ Que es esto? Bueno, existen tres instrucciones MOV que toman en cuenta la necesidad de código modificable; MOVS ("fuente" – source) solo almacena bits 0 al 8, MOVD ("destino" – dest) solo

almacena bits 9 al 17 y MOVI (“Instrucción”) solo almacena bits 23 al 3. Olvídate de poder usar estas instrucciones para manipular bytes ya que son para manipulación de 9 bits ☺ Pero algunos registros de I/O están organizados de una manera que puedes utilizar estas instrucciones.

Nota que: Existe una regla estricta: NUNCA modifiques la próxima instrucciones a ser ejecutada! Te explicare esto después en el articulo anexo F!

Te dije “…no ideada para manipulación de bytes..”, pero con la ayuda de Fred Hawkins diseñé esta pequeña joya:

```
'ex08A
 ‘Como empacar 4 bytes dentro de una celda del COG y escribir el valor en el HUB

  MOVI x, byte0     ' almacena en los 9 bits más significativos…
                    ' … dejando el “bit 31” como “no importa” (don’t care)
  SHR x, #8         ' empujar los bits a la derecha para liberar los 9 bits
  MOVI  x, byte1 ' repetir…
  SHR x, #8
  MOVI  x, byte2
  SHR x, #7          ' Atención! no empujes hacia afuera el bit menos significativo (LSB)
  SHR x, #1 WC    ' … pero mantén el valor en la bandera
  MOVI  x, byte3
  RCL x, #1       ' empuja el LSB : bit 31 era un bit “don’t care: pero ya no lo es
                  ' …
  WRLONG x, huba
```

Podría hacerlo más simple con este código:

```
‘Como empacar 4 bytes dentro de una celda del COG y escribir el valor en el HUB
'ex08B
  WRBYTE x, byte0
  ADD  x, #1
  WRBYTE x, byte1
  ADD  x, #1
  WRBYTE x, byte2
  ADD  x, #1
  WRBYTE x, byte3
```

Cuanto tiempo tomaría el ejemplo ex08B para ejecutar? Compáralo con ex08A!
pero espera! Y que acerca del “orden de los bytes”? en el ejemplo A el byte0 era el LSB y ahora es el MSB!

Bueno, no en realidad! Esto tiene que ver con lo que el Capitan Gulliver tuvo problemas: Pequeños Endians (Little Endians)! Un LONG en el HUB del Propeller es almacenado de manera que el Bit Menos Significativo (LSB) viene primero, y el MSB de último. No más comentarios, excepto que nunca notarás este comportamiento dentro de un COG, ya que no puedes desensamblar la estructura interna de las celdas.

Es posible “mejorar” el programa ex08B?
Claro que sí! Asuma que *byte0 al byte3* están contenidos en registros sucesivos:

```
byte0 long “a”
byte1 long “b”
byte2 long “c”
byte3 long “d”
```

```
‘Como empacar 4 bytes dentro de una celda del COG y escribir el valor en el HUB
'ex08C
   MOVD wrpatch:, #byte0 ' “implantar” la dirección del byte0 dentro de la instrucción
   MOV count, #4         ' preparar para almacenar 4 bytes en el bucle
:wrloop
:wrpatch
   WRBYTE 0-0, byte0     ' los 9 bits del “destino” serán remendados
   ADD  :wrpatch, aOneInDestPosition ' un remiendo muy audaz
   DJNZ count, #:wrloop


aOneInDestPosition LONG  %1_0_0000_0000
count  LONG 0
```

No es esto más elegante? Y – pon atención – no tomará más tiempo de ejecución que ex08B, ya que existe “tiempo utilizable” entre cada instrucción tipo WRxxx

Nota como audazmente evitamos problemas con la regla básica **“Nunca cambies la PROXIMA instrucción!”**

Y ni siquiera necesitamos más celdas que ex08B (=7) ☺

Articulo Anexo F: Como funciona la tubería de instrucciones (instruction pipeline)

Así que aprendiste que la mayoría de las instrucciones toman 4 ciclos de reloj (lo cual es 50 ns a 80 Mhz); inclusive si te dije eso, lo siento, pero eso es mentira!

Una instrucción “estándar” toma 6 ciclos:
T=0: Extraer instrucción
T=1: Decodificar instrucción
T=2: Extraer operando “destino”
T=3: Extraer operando “fuente”
T=4: Ejecutar operación
T=5: Almacenar resultado en “destino”

Como puedes ver la mayoría del tiempo pasa en el acceso a la memoria del COG (T=0,2,3,5), Así que puedes pensar, debería ser muy bueno poder saltar esos espacios de tiempo, si no existe un operando “fuente” que extraer, ya que tenemos una instrucción “inmediata” (T=3), o T=5, si no almacenamos nada de vuelta.

Pero el Propeller - al igual que otros procesadores avanzados – utiliza un método diferente para acelerar los procesos: “entrelaza” los accesos a la memoria de la PROXIMA instrucción entre los espacios de tiempo (T=1 y T=$) cuando la memoria no es usada para la instrucción ACTUAL, y así es como se ve:

Instrucción ACTUAL ULTIMA/PROXIMA instrucción

T=-1: Extrae "fuente" para la ÚLTIMA operación
T=0: Extrae Instrucción Ejecuta la ULTIMA operación
T=1: Decodifica instrucción almacena el resultado de vuelta en "destino"
T=2: Extrae operando “destino”
T=3: Extrae operando “fuente”
T=4: Ejecuta operación Extrae la PROXIMA instrucción
T=5: Almacena resultado dentro de "destino" Decodifica la PROXIMA instrucción
T=6: Extrae “destino” para la PROXIMA instrucción

No hay otra manera de hacerlo mejor!! La memoria es utilizada en cada ciclo de reloj; y una nueva instrucción es extraída cada 4 ciclos.

Ahora podrás entender, porque no se puede remendar “patch” la PROXIMA instrucción, ya que esta es extraída al T=4, mientras que el remiendo “patch” solo ocurre al T=5!

Y es importante mantener esta inter-temporización!

Sin embargo existen dos tipos de instrucciones que no pueden ser “temporizadas”, ya que toman una cantidad de tiempo indeterminada. Una es la familia de comandos WAIT, la temporización es algo como esto:

T=0: Extraer instrucción WAIT
T=1: Decodificar la instrucción
T=2: Extraer operando “destino”
T=3: Extraer operando “fuente”
T=3+N: Espera de cero a N ciclos
T=4+N: Almacenar resultado de vuelta en “destino”

Si no hay ninguna pausa, el comando toma 5 ciclos de reloj. No existe entrelazamiento de la PROXIMA instrucción; la extracción de la PROXIMA instrucción solo puede ser ejecutada al T=5+N

Espera! Una instrucción estándar toma 6 ciclos, una instrucción de WAIT que no hace pausa solo toma 5 ciclos, ¿ no debería tomar 3 ciclos solamente en el contexto del flujo de instrucciones? Muy astuto! Pero la naturaleza “variable” del WAIT (y la instrucción HUB) no está sincronizada en la tubería de instrucciones! La extracción de la nueva instrucción - la cual ocurre en T=4 para una instrucción estándar – no ocurre en T=3+N sino en T=5+N solamente – o así parece

La otra excepción es la familia de instrucciones de HUB. La temporización de un RDLONG es algo como esto:

T=0: Extraer instrucción HUB
T=1: Decodificar instrucción
T=2: Extraer operando “destino”
T=3: Extraer operando “fuente”
T=3+N: esperar de cero a 15 ciclos para que el HUB sincronice
T=4+N: direccionar el HUB
T=5+N: CARGAR/ALMACENAR desde/hacia el HUB
T=6+N: Almacenar resultado de vuelta en “destino”

También podemos tratar de entender la temporización de instrucciones de salto condicionales, El procesador siempre “predice” cuando va a ocurrir un salto y extrae la PROXIMA Instrucción desde esta dirección en T=4. Si la instrucción “continua la ejecución” entonces fue una mala predicción, y la extracción tiene que ser repetida en T=6. Sin embargo la intención en T=6 no es de ser un fase de “extracción”, ya que los saltos están sincronizados en la tubería de instrucciones (pipeline), en contraste con las instrucciones WAIT y HUB. Y la próxima “extracción estipulada” T=8

Note: el mecanismo presentado aquí no está muy bien descrito en la documentación oficial de Parallax y depende mayormente en mi propio “entendimiento”

Por cierto, has notado “RES”? in ex07C? esta es la ultima “directiva assembler” que “reserva” espacio de memoria pero sin almacenar datos, por lo que pensaras: No existe memoria HUB para estas celdas!. Cuando piensas un rato acerca de esto concluirás que esto no es posible, o solamente al final del programa con no preseteo o instrucciones que le sigan, y estas en lo cierto!

Capitulo 6 Bloqueadores (Locks) y Semáforos

Existe mucha preocupación acerca de esto: Realmente necesito “bloqueadores”? O “banderas”? pero por qué? Nunca las necesite antes!

Un hecho es que en cada sistema paralelo – ya sea hardware verdadero o a través de software – esto es mandatorio, aunque no para todas las aplicaciones, Así que semáforos binarios tienen que existir en el hardware del Prop, llamados “locks”

Considera el siguiente escenario:

Una tienda de departamentos tiene una entrada y una salida para poder controlar mejor el flujo de clientes diariamente. Ahora, la administración quiere saber cuántos clientes (o empleados) están en el edificio en cualquier momento del dia. Ellos piensan que pueden optimizar la asignación de empleados y cerrar la tienda con más confianza en la noche, Así que relés fotoeléctricos son instalados en la entrada y salida…

Generalmente existe dos tipos de respuestas para esta pregunta: Cuantas personas están dentro de la tienda?

```
ex09A
(- a la entrada-) IF signal THEN inCount += 1
(- a la salida -)   IF signal THEN outCount += 1
(- en la oficina -) personsInStore := inCount - outCount
```

Este enfoque tiene algunas desventajas”

- Ambos contadores pueden desbordarse
- El resultado siempre tiene que ser “calculado”, asi que realmente no esta disponible en línea “realmente”

Pero si esto no es problema entonces siempre preferirás esta solución!

```
ex09B
(- a la entrada-) IF signal THEN personsInStore += 1
(- a la salida -)   IF signal THEN personsInStore -= 1
(- en la oficina -) display (personssInStore)
```

Esta es probablemente una solución simple que viene a la mente, pero tiene una trampa!

Si el código- tal como es –NO ES ejecutado de manera serial, sino que tenemos unidades de procesamiento independientes cerca de la puerta, es posible que ambas accedan el "acumulador" *personsInStore* de manera contemporánea.

Por supuesto no existe tal cosa "contemporáneamente", pero nunca sabes y esto es de hecho un asunto filosófico de la digitalización de señales.

Asumamos ahora el técnico de la tienda es un Fan del Propeller ☺ y ejecuta "supervisión de la salida" en COGA y "supervisión de la entrada" en COGB

```
' department store ex09C
CON
  entPin = 2
  exPin  = 3
VAR
LONG personsInStore

PUB  main
  PersonsInStore := 0
  cognew(@entrance, @ personsInStore)
  cognew(@exit, @ personsInStore)

DAT
  ORG 0
entrance
  WAITPEQ :null, #entPin
  WAITPNE :null, #entPin
  RDLONG :e, PAR
  ADD :e, #1
  WRLONG :e, PAR
  JMP #entrance

:null LONG 0
:e    RES  1

  ORG 0        'note que este ORG es muy importante!!
exit
  WAITPEQ :null, #exPin
  WAITPNE :null, #exPin
  RDLONG :a, PAR
  SUB :a, #1
  WRLONG :a, PAR
  JMP #exit

:null LONG 0
:a    RES  1
```

Es obvio que tendrás problemas si te encuentras con la situación siguiente:

```
RDLONG a,..
ADD a, #1
RDLONG e,..
ADD e, #1
WRLONG a,..
WRLONG e,..
```

Uno de los clientes que esta entrando no será contabilizado. La probabilidad de esto es desconocida.

Pero, podemos eliminar este problema?

Ok, solo tenemos que encadenar las tres instrucciones RDLONG - ADD – WRLONG en una unidad irrompible (que no pueda ser interrumpida)!!

Procesadores estándar generalmente te dan una de estas (o ambas) soluciones:
- Deshabilitar Interrupts
- Una instrucción especial de “Leer-y-Modificar”

Cuando tienes hardware separado, solo la segunda opción es aplicable; la instrucción Leer-y-Modificar en el Propeller es llamada LOCKSET o LOCKCLEAR respectivamente.

Por supuesto no puede usar una celda arbitraria del HUB sino uno de los ocho bits específicos (llamados LOCKs)

Así que podemos mejorar nuestro código de la manera siguiente:

```
'ex09D

' entramos aquí cuando el relé envía la cenal
:l
   LOCKSET  sema  WC
   IF_C JMP #:l  ‘ espera hasta el que la persona salga de la zona critica

' ahora entramos en la zona critica
  RDLONG :a, PAR
  SUB :a, #1
  WRLONG :a, PAR
  LOCKCLEAR sema  ' aquí salimos de la zona critica
```

El nuevo programa modificado ahora luce de esta manera; también lo “mejoramos” un poco en algunos detalles para que el lector interesado se nutra con el conocimiento…

```
' department store ex09E
VAR
  LONG personsInStore

PUB  main

 PersonsInStore := 0
 semaNumber := LOCKNEW ' rserves a new semaphore
                  ' (Good code would check for < 0)
 countAddress := @ personsInStore

 pin := 2
 delta := 1
 COGNEW(@guard, 0)    ' Entrance
 WAITCNT(cnt+512*16)  ' it takes time to load a COG

 pin := 3
 delta := -1
 COGNEW(@guard, 0)    ' Exit

DAT
guard
  WAITPEQ :null, pin
  WAITPNE :null, pin
:w
  LOCKSET semaNumber WC
  IF_C JMP :w
  RDLONG a, countAddress
  ADD a, delta
  WRLONG a, countAddress
  LOCKCLR semaNummer

' debounce switch
  MOV a, :debounceTime
  ADD a, CNT
  WAITCNT a, #0
  JMP #guard

:null long 0
:debounceTime  LONG 80_000_000/1000 * 10  ' 10 ms
pin            LONG 0
delta          LONG 0
semaNumber     LONG 0
countAddress   LONG 0
a              RES  1
```

Another fascinating feature of the Prop is the easiness with which it can generate video signals. There is a little bit magic in the NTSC colour generation, but not much. We shall understand everything after we have worked ourselves through the following three chapters. A little bit lengthy – may be – but not really complex.

There is some specific hardware we shall call video logic in each COG, which is also handy for

- general 8-bit output
- especially if connected to a D/A converter of the R2R kind

This should be no surprise, as video is just an analogue signal as any other (or even three in the case of VGA: R, G, B)

A COG uses one of his timers ("A") to produce a certain clock to drive this video logic. This timer can be programmed in a wide range, thus allowing a wide range of applications! How to do this is explained in Parallax's excellent Application Note AN001. I have no intention to repeat the contents of it here. I do it for the German readers, but it would be folly to retranslate this! Just BTW – you English readers have no idea how privileged you are to understand the excellent Parallax documentation written in your mother language. Do use this privilege!

But we will see the timer working later in the examples of course.

Back to the "video logic": it has three modes of operation

- Composite Video (Baseband)
- Broadband Video ("TV")
- VGA

There are a few peculiarities in the former two that might confuse us in the beginning, so we start with the VGA mode. In this chapter we shall not yet wonder why it is called VGA – it's just a name!

So we set this mode – "Vmode = 01" - in **VCFG** ("Video configuration") - one of the two visible video registers; the other is called **VSCL**- "video scale". You will most likely have to look at the tables in the Propeller Datasheet (or manual) from time to time to find your way through the different fields in the 32-bit configuration register. It makes no sense to copy these tables here.

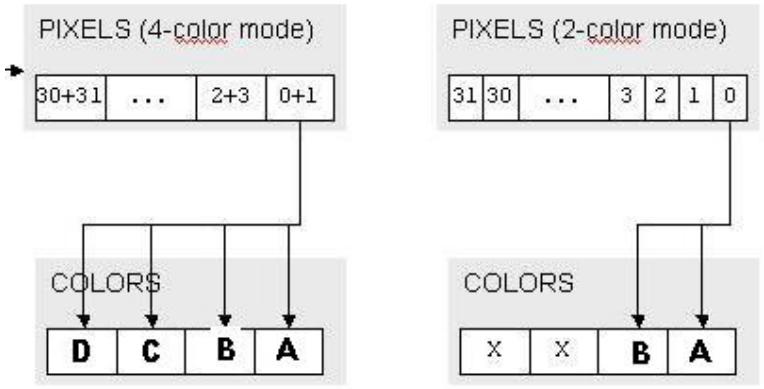
Whatever mode we have selected, there is not much difference for the rest of the handling.

There are also two hidden registers, I like to call PIXELS and COLORS (as I know of no official names): they must exist somewhere in the COG, though they cannot be read but just be set using the WAITVID instruction. This instruction in fact is the theme and centre of this chapter!

Let's imagine we have somehow connected the bespoken clock to the PIXELS register. PIXELS is a (arithmetic) "right-shift-register", and we are now shifting-out pixels, starting at the LSB end, either bit by bit, [b]or[/b] bit-pairs by bit-pairs. These are two different sub-modes called "2-color-mode" or "4-color-mode" for reasons that will become obvious later.

Now, what happens to these 32 out-shifted bits (or 16 out-shifted bit-pairs)? Where are they shifted to? Good question! To our great surprise, they are NOT shifted out of any I/O pin!

Rather, they are used to [b]address[/b] one of two bytes A or B (or one of four bytes A, B, C, or D) of the COLORS-register! Have a look at this Diagram 2:



Now, exactly one of these bytes A, B, C, or D is output at some configurable 8 pin group (I/O 0..7, 8..15, 16..23) Be careful with group 24..31 the use of which is also possible.

Well, this is nearly all to be said about Propeller video, except some minor details.

Did I already explain how to **set** PIXELS and COLORS? It is done by the WAITVID instruction. So if you want to output a specific bit pattern, just do:

```
WAITVID (eightbitPattern, 0)
          ^       ^
          |       |
        COLORS    PIXELS
```

Now 32 „times“ (the most right eight bits of) eightbitPattern is output. When you have selected the 4-color mode this happens only 16 “times”; but it will be difficult to spot the difference…

We call this sequence of output patterns a “register frame” (being of length 32 or 16)

Now we want to do something more adventurous: output a sequence of bytes like $FF, $1F, $07, $00; when we connect a R2R network to the eight pins it would look like a saw tooth at the end.

Using „4-color mode” this can be accomplished by:

```
WAITVID ( $FF_1F_07_00, %%0123 )
```

This is just ONE saw tooth followed by zero values; but we easily can generate 4 “saw teeth”:

```
WAITVID ( $FF_1F_07_00, %%0123012301230123 )
```

But there is a further configuration parameter („frame clock“, in the second video register **VSCL**), allowing to reduce the number of output patterns. Setting “frame clock” to 4 means that only the most right 4 (or 8) bits from the PIXELS are processed before the **WAITVID** instruction releases the video logic again; the “register frame” has now a length of 4 rather than 16.

This all sounds a little bit complicated .. What is the advantage when compared to some simple OUTA instructions?

Very little: It can work faster, and you do not have to take special care for timings and wait times. WAITVID has got its name for a very good reason. Before it starts shifting out the operands it waits for the end of a **previous video operation**! The **next** instruction is fetched as soon as the new shifting has been started. This is handy!

```
' ex10A
VSCLpreset   LONG  $1_004
VCFGpreset   LONG  %0_01_1_00_000_00000000000_0XX_0_11111111

'please do look up the meaning of all those parameters!!

colors       LONG $FF1F0700
   MOV VSCL, VSCLpreset
   MOV VCFG, VCFGpreset
LOOP
   WAITVID colors, #%%0123
   JMP #loop
```

This 4-byte saw tooth is output at pins XX*8 to XX*8+7 at most(!) every 8 clocks = 100 ns/4 bytes = 40 MB/s. This is not bad! Note that the limit is NOT the – still unknown – clocking of the video logic, but our ability to control this logic with instructions!

When we want to output more ambitious signals, we either have to compute them or load them from HUB; this slows things down further!

```
'ex10B
loop
  RDLONG colors, address
  WAITVID colors, #%%0123
  ADD address, #4
  DJNZ period, #loop
```

Though slowed down, this still yields 10 MB/s

But a “hand made” loop is not necessarily MUCH slower:

```
'ex10C
LOOP
  RDBYTE val, address
  SHL    val , #ioPinPos
  MOV    OUTA, val
  ADD    address, #1
  DJNZ   period, #loop
```

This is around 3 MB/s.

When we want later to generate video (4- 6 MHz) or true VGA (25-30 MHz), we already see that we are working at the frontier. Using a “register frame” of just 4 will not do; we shall need a frame length of 8, 16 or even 32 (in “2-color-mode”)

But why do I sound that this could be an issue? Why don't we use a "register frame" of 32 all the time in the first place?

Think! When shifting 32 bits in 2-color-mode the only option is to either output byte A or B from the COLOURS register: "Black" or "White" (or whatever you have stored there). The choice widens when using 4-color mode, but you are restricted to those patterns in the COLORS register for the whole register frame. This is severe constraint as soon as you use a frame size above 4!

Here is another extreme example, we use just **one** I/O pin rather than all 8

```
'ex10E
VSCL_preset   LONG  $1_020
VCFG_preset   LONG  %0_01_0_00_000_00000000000_0XX_0_00000001
colors        LONG  $01_00

  MOV VSCL, VSCL_preset
  MOV VCFG, VCFG_preset
loop
  RDLONG  data32, address
  WAITVID colors, data32
  ADD     address, #4
  DJNZ    length, #loop
```

We can output this bit stream with 32 bit per 400 ns, i.e. 80 Mbit/s per channel (or 40 Mbit/s per two channels in “4-color-mode”).

Someone may still wonder how to use the clock to drive the video logic. As seen above we can use the system clock for 80 MHz. It generally makes no sense to use a much higher clock rate, as we can no longer feed the video logic without disruptions; this would lead to unwanted “bursts” and “jitter”… We have to always adjust the video clock so that there will be a **small** wait left for a new WAITVID!

Sidetrack G: How to program Timer A

Programming timer A is done setting **CTRA** and **FRQA**, e.g.:

```
MOVI   FRQA, #56          '56*80/512 = about 8,75 MHz
MOVI   CTRA, #%00001_101  'internal,  PLL = *16/4 = *4 = 35 MHz
```

Need an explanation? All right: We use the [b]internal[/b] timer mode, so the timer signal (bit 31 of PHSA) is NOT output to any I/O pin but rather connected to shift bits out of the PIXELS register. This frequency is determined by the value of FRQA, which is added each system clock tick onto PHSA.

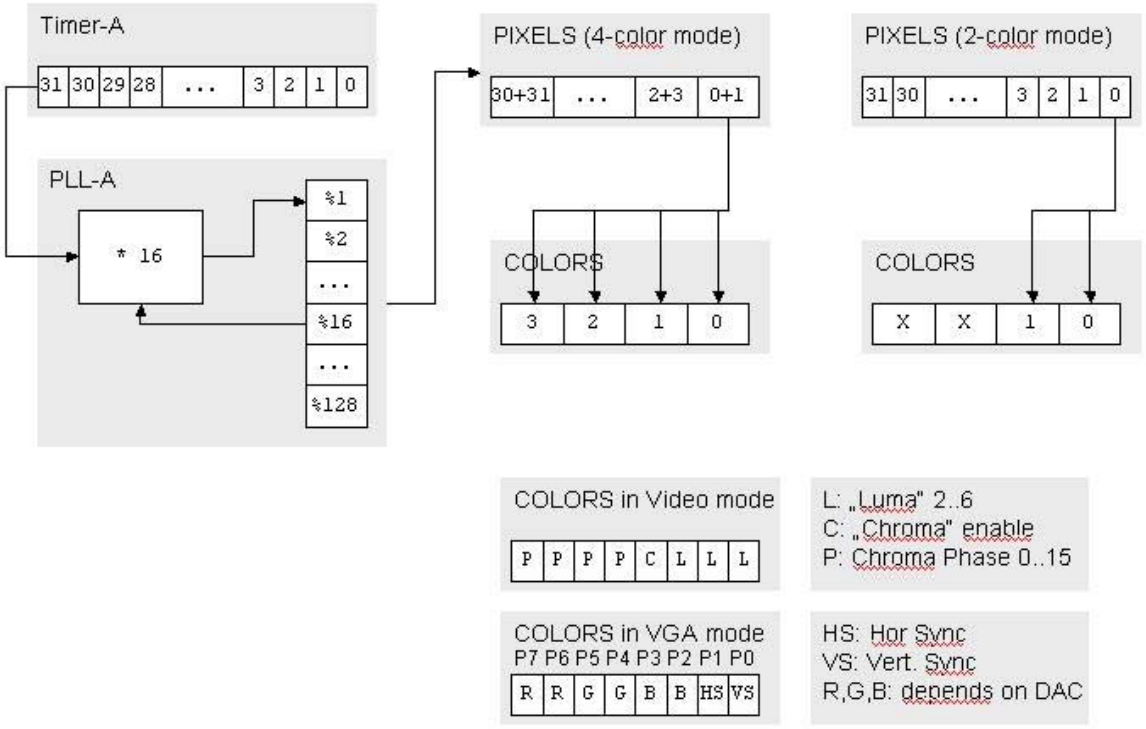
Logically this derived clock is always a fraction of the system clock, but this derived clock is used to control a PLL circuit multiplying the clock by 16! In addition to this boost, the PLL will compensate for any jitter if the fraction programmed for the timer is an “odd” number. PLLS work only within certain ranges: the datasheet says: 4 to 8 MHz (which means the clock at PLL output is 64 to 128 MHz).

However we will generally not use THAT clock, but a derived clock by dividing this frequency by 2 (32 – 64 MHz), 4, 8, 16, 32, 64, or 128 (500-1000 kHz).

To program a timer (i.e. the FRQA register) we often use the MOVI instruction, when the value can be “so la la” (1% off). Note that we have to calibrate the crystal and consider temperature anyway

when we want to do much better! A ONE in bit 23 results in an Timer overflow after 512 steps = 80 MHz/512 = 156 kHz. This is not good for the PLL (at least according to the datasheet – in practice it will do fine); 25 is the minimum value. The given value in ex10 (=56) will yield 8,75 MHz before we enter the PLL.

The next Diagram 3 will show these relations (timer and video logic) in a simple sketch; please ignore the 4 boxes in the lower right corner; they are needed in the next chapter only.



I have to apologize that there had been no complete examples up to now. The reason is, they would have made no sense. If you want to just copy code to see something happen you can as well use one of the many video drivers. But now you should have developed enough understanding of what is going on!

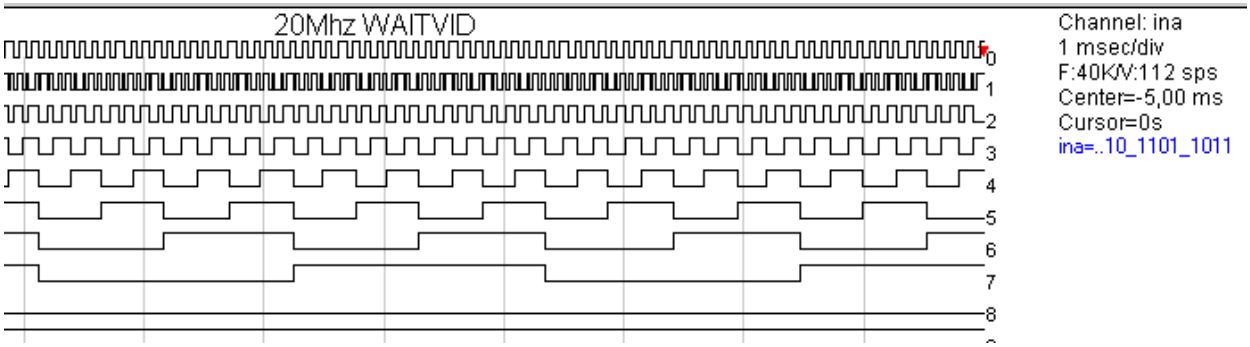
The name of this chapter was: “Video without Video”. No kidding: We will NOT have video now! Rather we will “count up” a set of 8 I/O pins (0..7), as slow as possible using the “video logic”. And don’t be surprised: as we are not interested in speed we can easily use SPIN for the next examples. (You know SPIN, don’t you??)

```
'ex11A
CON
  _clkmode = xtal1 + pll8x
  _xinfreq = 10_000_000

  _pinGroup = 0
  _pinMask = $FF<<(_pinGroup*8)

PUB  Main | n
   DIRA := _pinMask
   CTRA := %00001_000 << 23  ' internal, PLL % 128
   FRQA := POSX/CLKFREQ*4_000_000
```

```
  VSCL := $1_004          ' register frame = FOUR elements
  VCFG := %0_01_0_00_000_00000000000_000_0_11111111 +
        (_pinGroup << 9)  ' VGA, 2-color-mode
  REPEAT
    WAITVID (n++ ,0)
```

A cheap frequency counter will show us 240 Hz at pin 7 (MSB). My intention was to output just ONE element per frame (VSCL := $1_001), but the signal looked not very stable in that case. I think I have just spotted the reason: Just ONE element per frame has driven WAITVID to 240 Hz *4 *128 = 128 kHz corresponding to 8 us which is below the execution time of the SPIN loop!

As I just got crazy about the new **ViewPort** tool, I have to include a screenshot (Diagram 4):



But we can do much slower!

```
'edx11B
  VSCL := $ff_0ff  'now just 1 element/frame but stretched by 255
```

The frequency counter at pin 7 (MSB) now shows 4Hz (= 240/255*4)

In ex11A we shifted out 4 (equal) elements per register frame, which took the four-fold time. We could also shift out all 32 (equal) elements of PIXELS, but the field sizes in VSCL would not allow this (only 12 bits for "frame clock"); so we have to restrict ourselves to 16 elements.

```
'ex11C
  VSCL := $ff_ff0 '16 elements (frame length), 255-fold stretched
```

As expected the MSB (pin 7) now outputs at a quarter Hz

Last and least we shall challenge the slowest frequency of the timer the PLL will work with. I still succeeded at 65 kHz (the data sheet said: 4 MHz!)

```
'ex11D
  FRQA := POSX/CLKFREQ*65_000
```

There are few frequency counters showing anything at pin 7 now.
But an LED connected to pin 0 virtuously blinks in a two second rhythm.
(16*255*16/65_000 = 1 s)

Interlude 3: What Video is all about: A very gentle approach

Among recent high-tech devices a TV set is most likely the stupidest thing you can imagine. You have to tell it EVERYTHING in most detail! Not once, not twice, no, thirty times – each and every second! European TV sets are somewhat smarter; they only need to be reminded 25 times per second to their task.

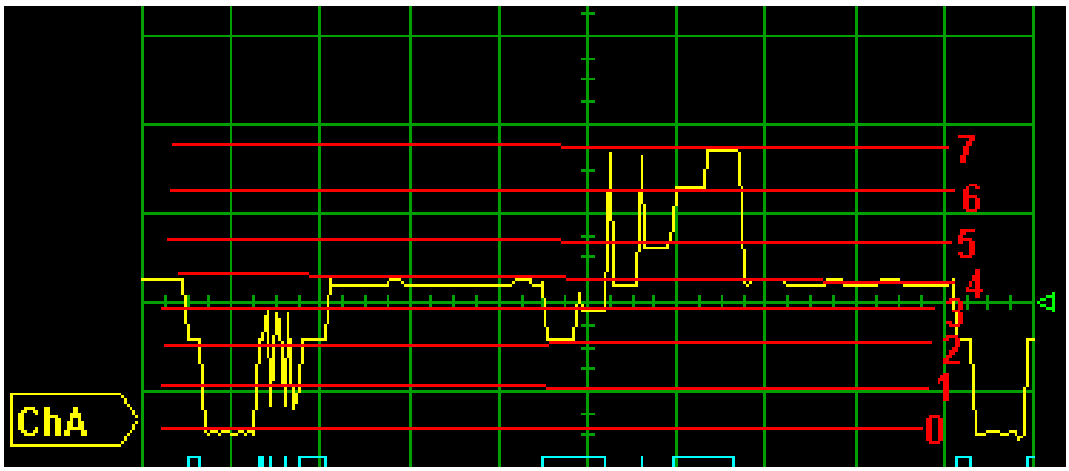
This is no coincidence: TV technology was invented somewhere between Morse's Telegraph and Bell's Telephone. The first TV standards where already established around 1928 (30 lines, 12.5 Hz screen refresh)

I can only think of one device stupider than a TV set: A (non multi-sync) VGA monitor.

But how do they work at all? Well in early times a beam of electrons was focussed on the other - more or less plane and transparent - end of a large radio tube. This end had been coated with some phosphorescent material that transmuted the electronic power into photonic light (Poetic – isn't it?) People were magically attracted by this effect as many are still today.

This beam took its way from the upper left corner to the lower right corner, covering the whole "screen" in exactly the way we read or write in most European languages.

During this path (which according to today's standards takes exactly 16.683 ms) its "luminosity" is changed, so a differentiated image can be displayed. The signal you will need to control the beam looks like the next Diagram 5:



(It is only a small part of the 16 ms – just one "line" of very few pixels and of just a few us. And thanks to Andy, who prepared it!)

Such kind of signal was originally directly applied to one of the grids of the tube.

In contrast to a popular prejudice, electrons rarely move with light speed. So when the beam returns from right to left its "beam power" had to be reduced as much as possible, down to "super black" so to speak; this phase is called "horizontal sync", followed by a "porch" until the beam has reached its new starting position at the left hand side. There has to be a similar gap in the signal when the beam is moving from bottom right to top left every 1/60 second.

When you want the Prop to produce "video" you have to generate exactly that kind of signal – nothing else at all, no feed back, no input, no test, no check, no if: It will be the most straightforward task in your career as a programmer. Of course you have to know the rules

There was no such thing as a “pixel” at that time. The size of the spots of light emitted from the screen depended on the tubes capability to “focus” the beam. In Y-direction the TV set had to care for the “spacing” of the line, so the number of lines had to be “implemented” in the devices and thus to be standardized early; in X-direction the bandwidth of the video amplifiers set a limit. The levels of the luminosity however stayed “analogue”.

The modern NTSC-M standard defines 720 horizontal and 486 vertical visible pixels.

With the advent of Colour TV “pixels” became more obvious to the public: for each of those formerly somewhat theoretical pixels three coloured dots had to be etched onto the tube. (Needed a lot of busy Chinese, I think!) Of course they couldn’t use less pixels than in the black and white standard for such an expensive device!

And the new Colour TV had to be “compatible” in both directions, that meant: a "monochrome" TV should not be disturbed by “colour signals”, and a Colour TV should be able to live happily without colour.

The solution for this can be considered as one of the magic moments of mankind!

When you look at the 'Diagram 3 you have little hints of colours (I shall try to find some more instructive images, alas my equipment does not allow me to catch those phantastic oscillograms myself at the moment). What you see is the “levels of grey”, the "luma"-signal. The ingenious idea now was to just add a small digital (!) signal – on the Prop adding one "level". However this signal was not locked in the framework of the “luma pixels”, but time shifted (also called: “phase” shifted) a little bit. The amount of this “jitter” or “displacement” was (and is!) used to define a “colour”: No displacement means “blue”, etc. Obviously the basic clock of this "chroma signal" has to be very precise, the value is: 3.579.545,00 Hz - and that's not kidding! Note the ",00"!

This specific feature made me say TV sets are somewhat more intelligent than VGA monitors are 

To generate this displacement of the "chroma signal" is a little bit tricky to accomplish, and this is the “little magic” going on in the COG’s video logic: You just say: “Blue”, and that’s is!

A black and white TV wouldn’t care, even wouldn’t notice this at all! A colour TV will have to make aware that those displaced signals have to be spotted for. But displaced to what reference? For this reason at the beginning of each line - in the midst of the “back porch” - an “awareness signal” is generated, called “colour burst”. The TV set will sync to it (using a PLL in the old days) to detect the displacements.

This “colour burst” however is NOT part of the “magic” – you have to generate it yourself in your home made driver…

There are more standards all over the world but NTSC-M: PAL and (the very similar) SECAM are most notable. As most small TV sets or car monitors are produced in countries using NTSC (No, I do NOT refer to the United States :-)), they understand this as their “mother language”. PAL is often an “add on”. It is different with the living-room TV. For the Propeller this (and “other formats” as 16:9) is most annoying. Much of the simplicity of video generation gets lost when you try to consider and care for multiple formats.

We also have to understand another more technical detail called “interlacing”. The alert reader has long noticed that deSilva has flopped again: "16 ms? That's not 1/30 second!"
Oh, dear! But you shall find enlightenment soon: The standard requires to transmit the screen contents 30 times per second (29.97 times to be very precise). Doesn’t that flicker? You bet it does! One could choose special screen coating for high afterglow (=”persistence”), as you most likely have seen on RADAR screens. But this is no real solution for fast changing images. A better solution was to split the screen lines (the "frame") into two groups – odd ("upper field") and even ("bottom field") lines – and transmit (and display) them one after another, each 243 double spaced lines taking 1/60 seconds. This uses afterglow “a little bit”, and most people now notice flicker only from the corner of their eye.

This had consequences for the producers of small and cheap monitors. They could successfully develop the attitude that only 243 lines matter and reduced the (expensive) TFT cells to 234 lines (it is unclear why exactly 234, maybe it started with a typo?). As this would result in very “un-square” pixels, they boldly also cut the 720 horizontal pixels in halves; 320 is used in most of those devices today, adapting to 16:9 formats increased this again, to 480 horizontal pixels.

This is what your Propeller displays to in many cases: 320x 234 or 480x 234.

Pictures look best when exactly adapting to this format. Be extremely careful with “interlacing”. It will reduce quality considerably on those monitors. Interlacing can improve quality on large living-room TV sets and some high end car monitors (> $100) sporting true 640x480 pixels. PC monitors will rarely have a video input. Frame catcher cards or USB frame catchers are tricky – you have to consult their manual, but not always with success…

The End

No, this is NOT the end!
DeSilva has many ideas how to continue:
- Best Practices
- Efficient Use of multiple COGs
- Time vs. Space
- Debugging with Ariba's PASD

Also there are still three half-bred sections missing, "The end of video", some remarks wrt the NR post-fix, and a sidetrack "Tricks with OUTB"

But he will have two evening classes about Propeller Programming in the next months, having to be prepared - and the material used there will not be in English…

When time is left, deSilva will re-activate his Multi-Prop project again: A "stackable" low-cost system for "number crunching". Photographs will follow…

Real Programmers don't use SPIN!? O yes, they do! SPIN is extremely handy for slow applications. However it has it's drawbacks and pitfalls also (and especially!) a machine programmer must be aware of:

Scope

-1) The rules are relatively simple: NO OVERLOADING! NO SHADOWING!

-2) There is no other possibility to access variables of other objects but using GETters and SETters, however they spend considerable time, in contrast to modern OOP design, where you find a tendency to offer them "for free" (i.e. compilers generate inline code).

-3) GETting an **address** is fine and in the spirit of SPIN as being a "structured assembly language"
This pointer needs not necessarily address an "array", but can point to any place in VAR or DAT space (but see the next section for further pitfalls)

Memory Allocation

-4) VAR variables are **resorted** by the compiler: LONGS first, follow WORDs, follow BYTES; unawareness of this can lead to deep frustration

-5) In contrast DAT variables are **padded** if appropriate!

-6) Never forget: VAR is "object space"; only DAT is "global"!

Tree of Objects

-7) Each time you define a name in the OBJ section a new object is "instantiated". That means a new set of VAR memory is (statically) allocated. DAT and CODE always stays the same.

This is extremely frustrating when you have "library objects" used from multiple spots of your program. Take **Float32**. You may need it from the main object and some "sub-object" (maybe **FloatString**). FloatString normally uses the independent and slow FloatMath; so you are inclined to change that to Math32 as well. This is where your problems start ☺ But after you understand their root, you can easily fix Math32 (2 variables in VAR -> DAT)

-8) There is a bug – at least according to my opinion – in COGNEW, as it does not deliver the object context to the SPIN Interpreter in the new COG, which means you can only use procedures from your own object.

```
'Example
  OBJ
     Sub1 : "sub1"
  PUB main
     COGNEW (sub1.go(0),@stack)
'Does not work, and there is no warning
```