uc3m

Departamento Tecnología Electrónica

# Tema 4: Lenguaje Ensamblador

## Sistemas Digitales Basados en Microprocesadores

Universidad Carlos III de Madrid

Dpto. Tecnología Electrónica

# Índice

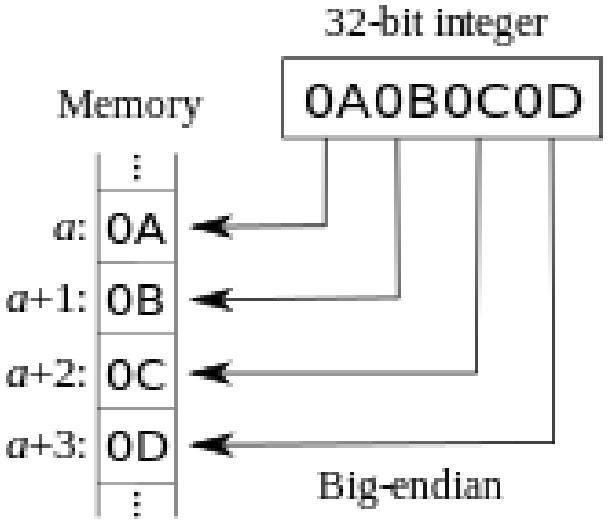
# 1 - Modelo de Programador

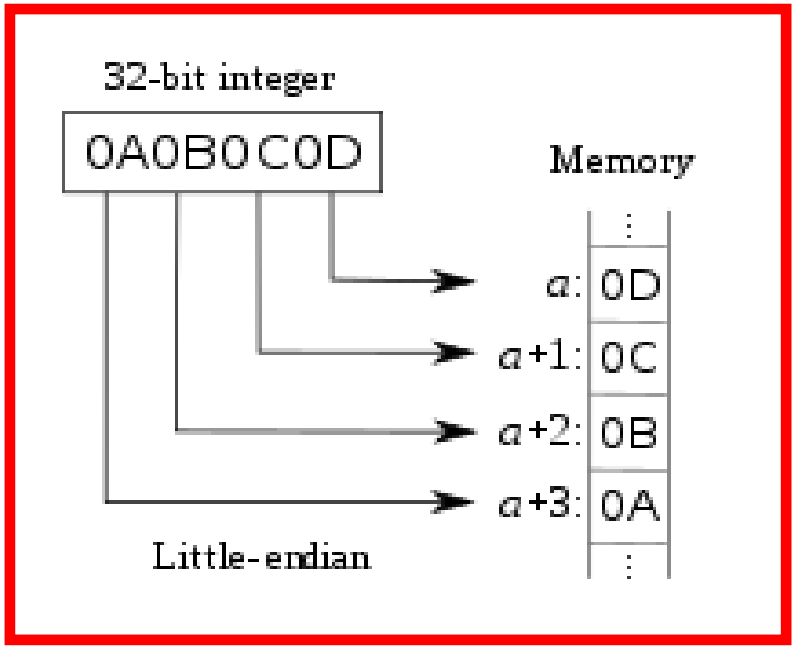
# Modelo del Programador de un micro

- A la hora de programar una CPU, muchos elementos internos se ocultan (por ejemplo registros como el MAR, MBR, IR, etc.), es decir, no son accesibles por parte del programador.
- Por esto y por más cosas, hace falta un Modelo de Programador que establezca el conjunto de elementos que son necesarios conocer de la arquitectura interna de la CPU para realizar programas. Son los siguientes que veremos en este tema:
  - Modelo de memoria (en este punto 1)
  - Tipos de Datos (en este punto 1)
  - Modos de Operación (en este punto 1)
  - Registros accesibles (en este punto 1)
  - Mecanismos de Programación (en el punto 2)
  - Juegos de instrucciones (en el punto 3)
  - Modos de direccionamiento (en el punto 4)

# Modelo del Programador en el ARM7

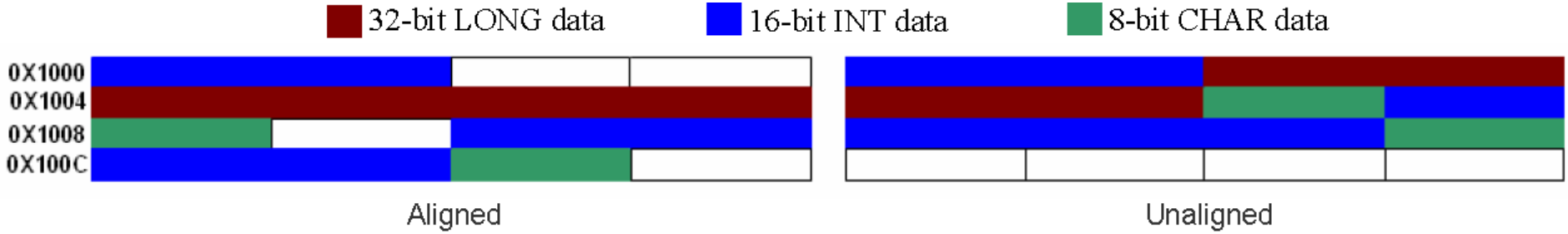
- Modelo de Memoria:
  - El AMR7 direcciona bytes (es decir sus 32 bits de direcciones direccionan datos con un ancho de 8 bits)
  - Pero como los datos pueden ser de hasta 32 bits, se utiliza una codificación little-endian, es decir, cómo se guardan los datos en memoria consecutivamente cuando son de 16 o de 32 bits (ya que la memoria sólo tiene un ancho de 8 bits)



# Modelo del Programador en el ARM7

- Tipos de datos:
  - El ARM7 puede trabajar con 3 tipos de datos:
    - Palabra (***word***) de 32 bits, que en memoria debe estar colocada alineada en múltiplos de 4
    - Semipalabra (***halfword***) de 16 bits, que en memoria debe estar colocada alineada en múltiplos de 2
    - ***Byte***, que en memoria puede colocarse en cualquier dirección
  - Además de los 3 tipos mencionados anteriormente, el ARM7 puede gestionar que cualquiera de ellos sea:
    - Con signo
    - Sin signo
  - Estos datos se pueden almacenar de forma alineada o no alineada en la memoria, aunque esto lo elige el compilador utilizado, no el programador. Un ejemplo de esto se ve a continuación.



# Modelo del Programador en el ARM7

- Modos de Operación:
  - Thread mode: para ejecutar aplicaciones software. Es el modo en el que entra la CPU cuando sale del estado de reset
  - Handler mode: para gestionar las excepciones. El procesador vuelve al modo Thread cuando ha terminado de procesar la excepción

  **-> Niveles de privilegio para la ejecución de software:** Esto se puede elegir en los micros modernos aunque en nuestro micro siempre tendremos nivel "Privileged"
    - Unprivileged: en este nivel, el software:
      - Tiene acceso limitado a algunas instrucciones
      - No puede acceder al reloj del sistema, al NVIC o al bloque de control del sistema
      - Puede tener acceso restringido a memoria o periféricos
    - Privileged: el software tiene acceso a todas las instrucciones y todos los recursos

- Registros accesibles:
  - Como ya se ha comentado, a la hora de programar una CPU, muchos elementos internos se ocultan (por ejemplo registros como el MAR, MBR, IR, etc.), es decir, no son accesibles por parte del programador. En nuestro micro no nos preocupamos de esto y ya veremos qué es accesible y qué no al ver los temas de programación.

# 2 - Mecanismos de Programación

# Mecanismos de Programación

- El desarrollo de un programa se realiza mediante la concatenación de instrucciones que se ejecutarán siguiendo un flujo lineal
- Sin embargo, ese flujo lineal se puede alterar mediante:
  - Saltos
  - Subrutinas
  - Interrupciones
- **Saltos:**
  - El programa continúa la ejecución en otro punto de la memoria y no tiene porqué volver al punto desde el que salta, como si lo veremos en las subrutinas
  - Se provoca mediante una instrucción de salto condicional o incondicional, en la que se indica la dirección a la que saltar
  - En el salto condicional, la instrucción del ARM7 se denomina B{cond} y si se cumple la condición, esa instrucción realiza el cambio del valor del PC por el de la posición donde tiene que seguir el programa.

# Mecanismos de Programación

- **Subrutinas:**
  - Se trata de un salto temporal, volviendo posteriormente a la instrucción siguiente a la del salto a subrutina
  - Se utilizan para estructurar el programa en funciones a las que se llama repetidas veces (optimizan código)
  - La CPU obtiene la dirección de salto de los bits que acompañan al opcode en la instrucción
  - Antes de realizar el salto, la CPU debe guardar el contenido del PC, para recuperarlo posteriormente (cuando se dé la instrucción de volver)
  - Las instrucciones para ejecutar subrutinas en el caso del ARM7 se denominan BL{cond} (*branch with link*), y guardan el valor del PC (R15) en el registro Link LR (R14)

# Mecanismos de Programación

- **Interrupciones:**
  - Es un mecanismo por el cual se permite que la CPU realice determinadas operaciones, cuando haya ocurrido un determinado evento externo
  - Es como una subrutina, pero que no es llamada por el programador (no hay una instrucción concreta en el programa)
    - Cada vez que se ejecuta, el programa principal puede estar en ese momento en cualquier instrucción
  - Cuando ocurre ese evento se pasa a ejecutar una subrutina, conocida como **Rutina de Atención a la Interrupción** o **Rutina de Servicio** (conocida como **RAI**)
    - Antes de entrar a ejecutar la RAI, la CPU tiene que guardar el valor del PC y el contexto (que en su versión mínima es el SR)
      - En algunas CPUs también se inhiben las interrupciones para evitar interrumpir la interrupción que se está ejecutando
    - Cuando se termina la RAI se recupera el contexto (el valor del PC y del registro de estado) y se sigue ejecutando el programa principal en la instrucción donde se quedó

# Mecanismos de Programación

- **Interrupciones:**
  - La RAI es algo que se puede ejecutar en cualquier momento, y que está interrumpiendo el curso normal del programa
    - Entre la CPU y el programador se debe garantizar que, cuando se vuelve de la RAI, la CPU se encuentra en las mismas condiciones que si no se hubiese producido la interrupción (salvo modificaciones intencionadas)
    - Debe ser una rutina razonablemente corta y efectiva, que no deje al programa en una espera demasiado larga (un cuelgue)
    - Tradicionalmente tendrá que comunicar al programa principal que ha ocurrido una interrupción y las consecuencias de la misma (usando variables auxiliares)
  - Una CPU puede tener varias fuentes de interrupción, y pueden estar activas más de una al mismo tiempo
    - Algunas estarán activadas por defecto, mientras otras habrá que activarlas (ya lo veremos en la parte de programación)

# 3 - Juego de Instrucciones

# Juego de Instrucciones del ARM7

- El Juego de Instrucciones del ARM7 es tipo RISC, es reducido en número de instrucciones, pero muy amplio en variantes de las mismas
  - Por ejecución condicional
  - Por modo de direccionamiento y tamaño de la palabra utilizada
- La complejidad del Ensamblador de una CPU hace que para este curso se realicen una serie de simplificaciones.
  - De esta forma se facilita un conocimiento general que le otorga al alumno la capacidad futura de analizarlo con detalle. Se van a seguir las siguientes pautas en el curso para las explicaciones:
    - Se comentará primero las distintas instrucciones
    - Después se detallarán:
      - Las condiciones de ejecución para las mismas
      - La forma de referirse a los distintos operandos (direccionamiento)

# Simplificaciones en el Curso

- Todas las instrucciones pueden ser condicionales
  - Esto se refleja en que cada mnemónico de instrucción va seguido opcionalmente por un código de condición {cond}
  - **Pero durante este curso no se van a utilizar los códigos de condición en ninguna instrucción salvo en las de Salto Condicional**
    - Por tanto se ignorará en todas la parte {cond} salvo en la instrucción B{cond}
- Muchas instrucciones pueden determinar si se modifica o no el registro de estado
  - **En este curso se va a considerar que todas aquellas instrucciones que puedan modificar el registro de estado, lo van a hacer**
    - Toda instrucción que pueda utilizar el parámetro {S}, se le pondrá de forma fija (MOVS, ADDS, etc.) para indicar que van a modificar el registro de estado
- Sobre la posibilidad de que una instrucción pueda realizar más de una operación (tal como, por ejemplo, un desplazamiento unido a una suma), esto **SI** que se va a utilizar, y por tanto no se simplificará el formato del <op2> (se verá más adelante en los ejemplos).
- La utilización avanzada de la Arquitectura Interna (gestiones de registros de estado, salvaguardas, pilas, etc.) se va a simplificar al máximo.

# Tipos de Instrucciones

- Hay que tener en cuenta que hay fundamentalmente dos tipos de instrucciones:
  - Transferencia de datos con memoria: utilizará un operando que llamaremos <am2>, <am3>, <am4> y <am5> (en este curso sólo <am2>), los cuales están relacionados con cómo se indica la dirección de memoria (esto se verá más adelante).
  - De tipo general: el segundo operando puede tener muchas variantes, por lo que normalmente se va a denotar como <op2> y nunca será una posición de memoria.
- Las instrucciones se van a desglosar en:
  - Transferencia de Datos
    - Entre memoria y CPU
    - Entre registros
  - Aritméticas y Lógicas
  - Control de Flujo
  - Misceláneas (no se van a ver en este curso)
- Los desplazamientos y rotaciones no son instrucciones específicas, sino una forma de tratar uno de los operandos (<Op2>) (se verá más adelante en los ejemplos).

# Transferencia de Datos con Memoria

| Mnemónico | Función | Sintaxis | RTL | N | Z | C | V |
|---|---|---|---|---|---|---|---|
| **LDR (Load Register)** | Carga una *word* | LDR{cond} Rd, <am2> | Rd ⬅ $(\text{<am2>})_{32}$ | - | - | - | - |
| | Carga un *byte* | LDR{cond}B Rd, <am2> | Rd ⬅ 000000:$(\text{<am2>})_{8}$ | - | - | - | - |
| | Carga un *byte* con signo | LDR{cond}SB Rd, <am3> | Rd ⬅ $(\text{<am3>})_{8,sign\_ext32}$ | - | - | - | - |
| | Carga una *halfword* | LDR{cond}H Rd, <am3> | Rd ⬅ 0000:$(\text{<am3>})_{16}$ | - | - | - | - |
| | Carga una *halfword* con signo | LDR{cond}SH Rd, <am3> | Rd ⬅ $(\text{<am3>})_{16,sign\_ext32}$ | - | - | - | - |
| **LDM** | Carga múltiples registros | *Hay 7 variantes* | | - | - | - | - |
| **STR (Store Register)** | Graba una *word* | STR{cond} Rd, <am2> | $(\text{<am2>})_{32}$ ⬅ Rd | - | - | - | - |
| | Graba un *byte* | STR{cond}B Rd, <am2> | $(\text{<am2>})_{8}$ ⬅ $Rd_{7..0}$ | - | - | - | - |
| | Graba una *halfword* | STR{cond}H Rd, <am3> | $(\text{<am3>})_{16}$ ⬅ $Rd_{15..0}$ | - | - | - | - |
| **STM** | Graba múltiples registros | *Hay 6 variantes* | | - | - | - | - |

# Transferencia de Datos entre Registros

| Mnemónico | Función | Sintaxis | RTL | N | Z | C | V |
|---|---|---|---|---|---|---|---|
| **MOV** | Mueve un dato | MOV{cond}{S} Rd, <op2> | Rd ⬅ <op2> | x | x | x | - |
| **MVN** | Mueve el negado del dato | MVN{cond}{S} Rd, <op2> | Rd ⬅ !<op2> | x | x | x | - |
| **PUSH** | Mete registros en la pila | PUSH{cond} Registro | SP ⬅ SP-4<br>(SP) ⬅ Registro | - | - | - | - |
| **POP** | Saca registros de la pila | POP{cond} Registro | Registro ⬅ (SP)<br>SP ⬅ SP+4 | - | - | - | - |
| **MRS** | Mueve el SPSR a un Rd | MRS{cond} Rd, SPSR | Rd ⬅ SPSR | - | - | - | - |
| | Mueve el CPSR a un Rd | MRS{cond} Rd, CPSR | Rd ⬅ SPSR | - | - | - | - |
| **MSR** | Carga el SPSR ({f} indica que grupo/s se actualizan: c=[7:0], x=[15:8], s=[23:16], f=[31:24]) | MSR{cond} SPSR_{f}, Rm | SPSR ⬅ Rm | - | - | - | - |
| | | MSR{cond} SPSR_{f}, #im | SPSR ⬅ #im | - | - | - | - |
| | Carga el CPSR | MSR{cond} CPSR_{f}, Rm | CPSR ⬅ Rm | x | x | x | x |
| | | MSR{cond} CPSR_{f}, #im | CPSR ⬅ #im | x | x | x | x |

# Aritméticas

| Mnemó. | Función | Sintaxis | RTL | N | Z | C | V |
|---|---|---|---|---|---|---|---|
| **ADD** | Suma | ADD{cond}{S} Rd, Rn, <op2> | Rd ⬅ Rn + <op2> | x | x | x | x |
| **ADC** | Suma con acarreo | ADC{cond}{S} Rd, Rn, <op2> | Rd ⬅ Rn + <op2> + C | x | x | x | x |
| **SUB** | Resta | SUB{cond}{S} Rd, Rn, <op2> | Rd ⬅ Rn - <op2> | x | x | x | x |
| **SBC** | Resta con acarreo | SBC{cond}{S} Rd, Rn, <op2> | Rd ⬅ Rn - <op2> - !C | x | x | x | x |
| **RSB** | Resta (inversa) | RSB{cond}{S} Rd, Rn, <op2> | Rd ⬅ <op2> - Rn | x | x | x | x |
| **MUL** | Multiplica | MUL{cond}{S} Rd, Rm, Rs | Rd ⬅ Rm * Rs | x | x | x | x |
| **MLA** | Multiplica y Acumula | MLA{cond}{S} Rd, Rm, Rs, Rn | Rd ⬅ Rm * Rs + Rn | x | x | ? | x |
| **SDIV** | Divide con signo | SDIV{cond}{S} Rd, Rn, Rm | Rd ⬅ Rn / Rm | x | x | x | x |
| **UDIV** | Divide sin signo | UDIV{cond}{S} Rd, Rn, Rm | Rd ⬅ Rn / Rm | x | x | x | x |
| **UMULL** | Multiplica unsigned long | UMULL{cond}{S} RdLo, RdHi, Rm, Rs | RdHi:RdLo ⬅ Rm * Rs | x | x | ? | ? |
| **UMLAL** | Multiplica y Acumula unsigned long | UMLAL{cond}{S} RdLo, RdHi, Rm, Rs | RdHi:RdLo ⬅ Rm * Rs + RdHi:RdLo | x | x | ? | ? |
| **SMULL** | Multiplica signed long | SMULL{cond}{S} RdLo, RdHi, Rm, Rs | RdHi:RdLo ⬅ Rm * Rs | x | x | ? | ? |
| **SMLAL** | Multiplica y Acumula signed long | SMLAL{cond}{S} RdLo, RdHi, Rm, Rs | RdHi:RdLo ⬅ Rm * Rs + RdHi:RdLo | x | x | ? | ? |

# Lógicas

| Mnemó. | Función | Sintaxis | RTL | N | Z | C | V |
|---|---|---|---|---|---|---|---|
| **AND** | And lógico | AND{cond}{S} Rd, Rn, <op2> | Rd ⬅ Rn & <op2> | x | x | x | - |
| **EOR** | Or exclusivo | EOR{cond}{S} Rd, Rn, <op2> | Rd ⬅ Rn ^ <op2> | x | x | x | - |
| **ORR** | Or lógico | ORR{cond}{S} Rd, Rn, <op2> | Rd ⬅ Rn \| <op2> | x | x | x | - |
| **BIC** | Bit clear | BIC{cond}{S} Rd, Rn, <op2> | Rd ⬅ Rn & !<op2> | x | x | x | - |
| **ORN** | Or lógico negado | ORN{cond}{S} Rd, Rn, <op2> | Rd ⬅ Rn \| !<op2> | x | x | x | - |

# Control de Flujo

| Mnemó. | Función | Sintaxis | RTL | N | Z | C | V |
|---|---|---|---|---|---|---|---|
| **CMP** | Compara | CMP{cond} Rn, <op2> | Rn - <op2> | x | x | x | x |
| **CMN** | Compara con negado | CMN{cond} Rn, <op2> | Rn + <op2> | x | x | x | x |
| **TST** | Test | TST{cond} Rn, <op2> | Rn & <op2> | x | X | x | X |
| **TEQ** | Test de equivalencia | BIC{cond} Rn, <op2> | Rn ^ <op2> | x | x | x | x |
| **B** | Branch | B{cond} etiqueta | PC ⬅ PC + <dir_rel de etiqueta> | - | - | - | - |
| **BL** | Branch con Link | BL{cond} etiqueta | LR ⬅ PC;<br>PC ⬅ PC + <dir_rel de etiqueta> | - | - | - | - |
| **BX** | Branch y cambio de juego de instrucciones | BX{cond} Rn | T ⬅ Rn[0];<br>PC ⬅ Rn & 0xFFFFFFFE | - | - | - | - |

# Códigos para la Ejecución Condicional

| Suffix | Flags | Meaning |
|---|---|---|
| EQ | Z = 1 | Equal |
| NE | Z = 0 | Not equal |
| CS or HS | C = 1 | Higher or same, unsigned ≥ |
| CC or LO | C = 0 | Lower, unsigned < |
| MI | N = 1 | Negative |
| PL | N = 0 | Positive or zero |
| VS | V = 1 | Overflow |
| VC | V = 0 | No overflow |
| HI | C = 1 and Z = 0 | Higher, unsigned > |
| LS | C = 0 or Z = 1 | Lower or same, unsigned ≤ |
| GE | N = V | Greater than or equal, signed ≥ |
| LT | N != V | Less than, signed < |
| GT | Z = 0 and N = V | Greater than, signed > |
| LE | Z = 1 and N != V | Less than or equal, signed ≤ |
| AL | Can have any value | Always. This is the default when no suffix is specified. |

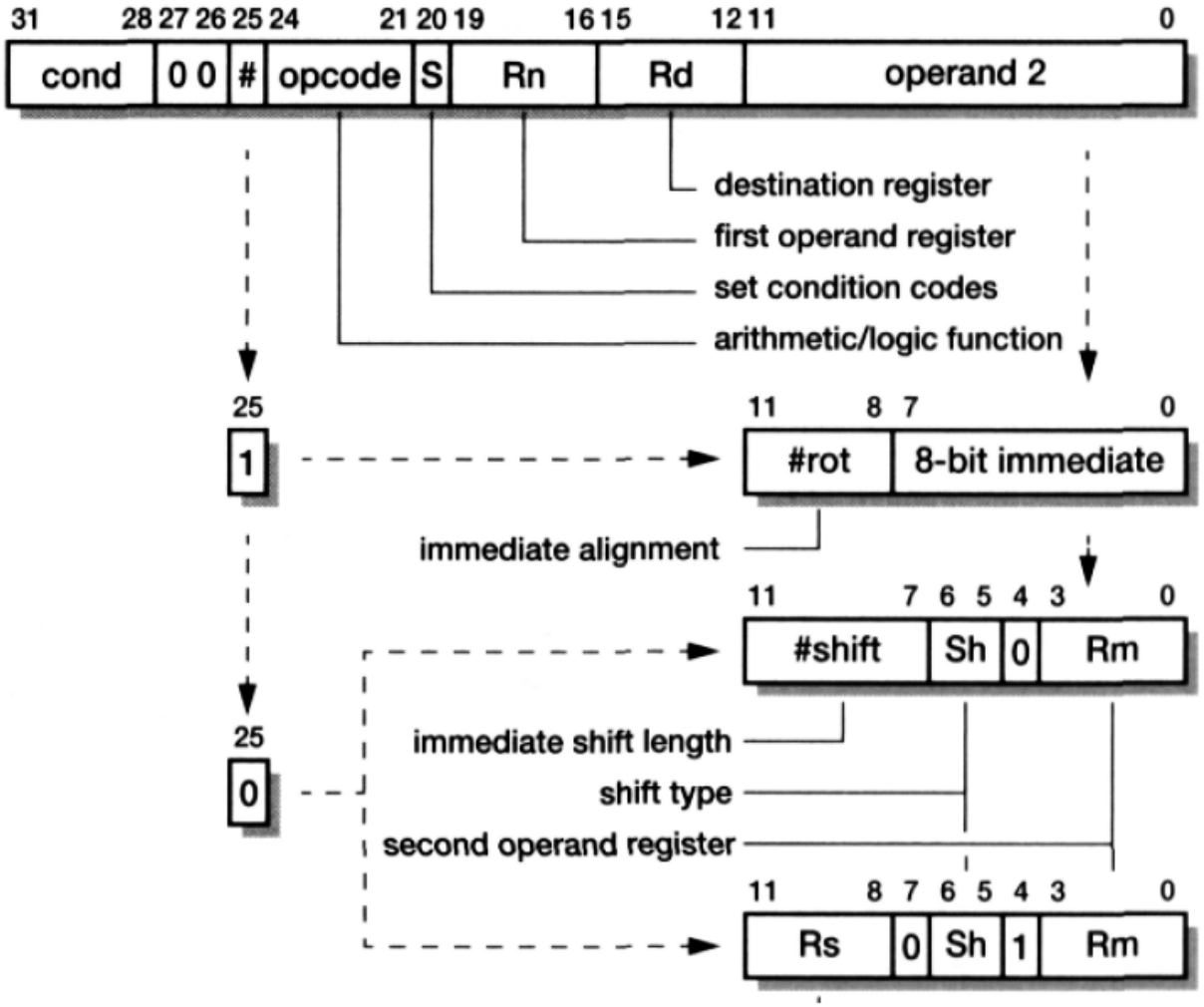
# 4 - Modos de Direccionamiento del ARM7

# Modos de Direccionamiento del ARM7

- La documentación del ARM7 suele hacer una clasificación de sus modos de direccionamiento, atendiendo a su momento de uso:
  - **Grupo 1 <op2>: Especifica el segundo operando en las instrucciones de procesamiento de datos**
  - En las operaciones de transferencias de datos a/desde memoria, el operando que no es un registro se puede dar de las siguientes formas:
    - **Grupo 2 <am2>: En operaciones de tamaño word o unsigned byte**
    - Grupo 3 <am3>: En operaciones de tamaño halfword o signed byte
    - Grupo 4 <am4>: En operaciones de transferencia múltiple
    - Grupo 5 <am5>: En operaciones relativas a co-procesadores

**-> ¡¡En este curso, como ya se comentó, sólo se tratan las marcadas en negrita!!**

# Grupo 1 con <op2>: Estructura



# Grupo 1 con <op2>: Explicación de la estructura

- Se toma como ejemplo la instrucción **ADD Rd, Rn, <op2>** (Son las del tipo de transferencia de datos entre registros, aritméticas, lógicas y de control de flujo de las transparencias 18-21)
- El operando <op2> puede ser una constante o un registro con un desplazamiento opcional. Vemos las diferentes opciones de direccionamiento para <op2>:
- **Inmediato** (ADD Rd, Rn, #inmediato)
  - Bit 25 = 1 y el dato del operando 2 es directamente un número de 8 bits colocado en los bits 0-7, siendo #rot = 0

    Ejemplo: ADD R1, R2, #25 -> R1⬅ Dato en R2 + 25
  - Pero este número se puede desplazar a la izquierda, según lo indicado en #rot (bits 8-11), el número de veces indicado en #rot (es decir de 1 a 15, lo que implica multiplicar por 2, 4, 8…. dicho número), con lo que podemos aumentar el dato más del número 256 (8 bits sin rotación)

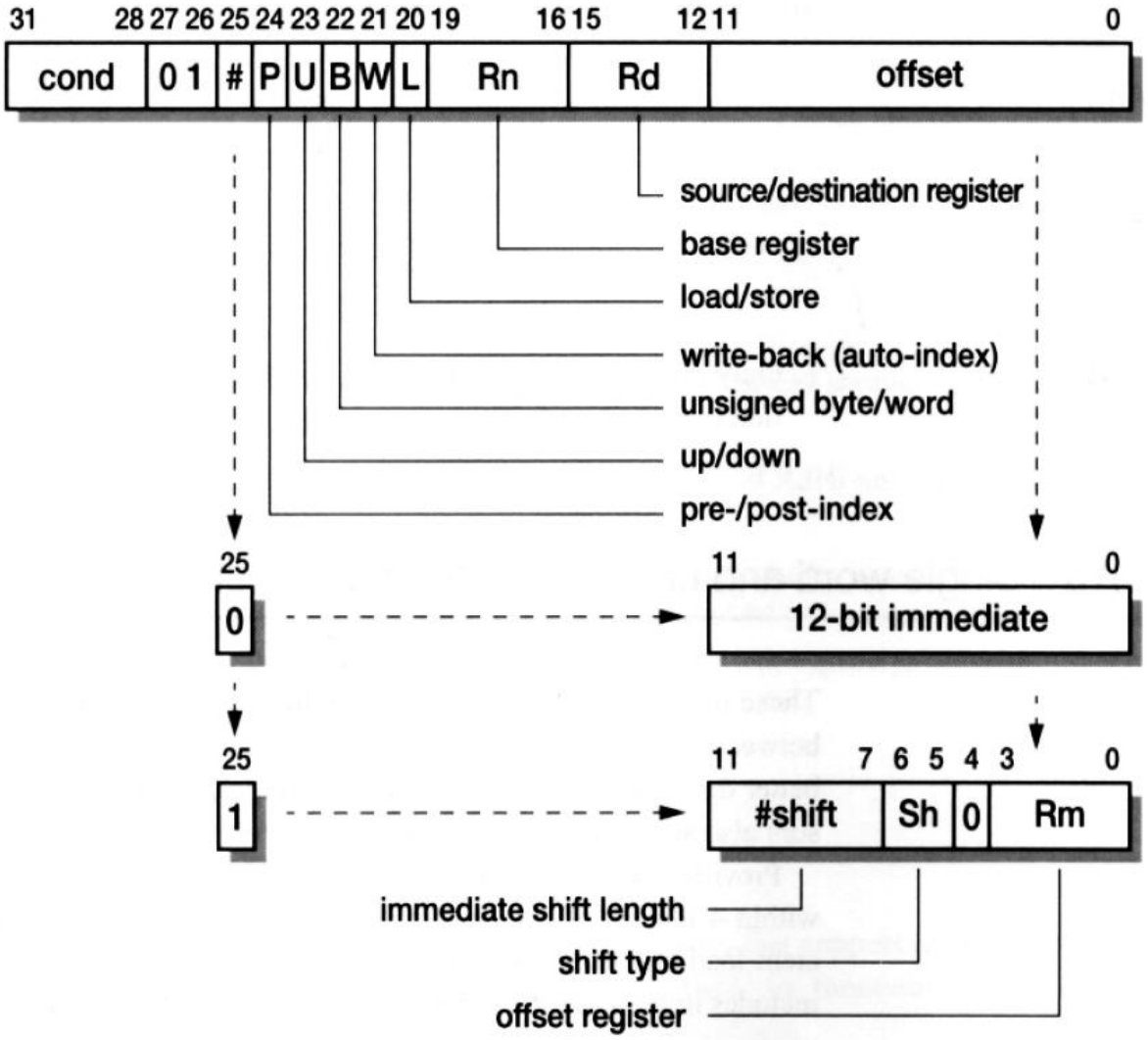
    Ejemplo: ADD R1, R2, #511 -> R1⬅ Dato en R2 + 510 / ($511_d$=$1FF_h$) -> el dato se desplaza 1 posición a la derecha y se multiplica 2 quedando $111111110_b$ = $510_d$
- **Directo a Registro** (ADD Rd, Rn, Rm)
  - Bit 25 = 0 y el dato del operando 2 está en un registro colocado en los bits 0-3, siendo #shift = 0 y SH = 0

    Ejemplo: ADD R1, R2, R3-> R1⬅ Dato en R2 + Dato en R3
- **Directo a Registro, con el contenido del registro desplazado**
  - Bit 25 = 0 y el dato del operando 2 se obtiene haciendo la operación SH entre el dato que hay en registro indicado en los bits 0-3 y lo que se indique en #shift o RS
    - La operación SH puede ser:
      - ASR – desplazamiento aritmético a derechas
      - LSL – desplazamiento lógico a izquierdas
      - LSR – desplazamiento lógico a derechas
      - ROR – rotación a derechas
      - RRX – rotación a derechas de un único bit, introduciendo por la izquierda C
    - El número de bits a desplazar o rotar en la operación puede ser (salvo para RRX):
      - Un número inmediato indicado en #Shift (ADD Rd, Rn, Rm, SH #Shift)

        Ejemplo: ADD R1, R2, R3 LSL #2 -> R1⬅ Dato en R2 + (Dato en R3*4)
      - Un número indicado en registro RS (ADD Rd, Rn, Rm, SH Rs)

        Ejemplo: ADD R1, R2, R3 LSL R4 -> R1⬅ Dato en R2 + (Dato en R3* $2^{\text{Número indicado en R4}}$)
- **<u>Importante:</u>** Si S=1, entonces se actualiza el registro de estado. Si S=0 no se actualiza el registro de estado

# Grupo 2 con <am2>: Estructura



# Grupo 2 con <am2>: Explicación de la estructura

- Se toma como ejemplo la instrucción **LDR Rd, <am2>** (Son las del tipo transferencia de datos con memoria LDR y STR de la transparencias 17)
- El operando <am2> puede ser una constante o un registro con un desplazamiento opcional. Vemos las diferentes opciones de direccionamiento para <am2>:
- **Indirecto con Desplazamiento** (LDR Rd, [Rn, #+/-<offset_12>])
  - Bit 25 = 0 y el desplazamiento es un número de 12 bits colocado en los bits 0-11. La dirección efectiva donde está el dato 2 se obtiene sumándole o restándole a la dirección base (dada por Rn) el desplazamiento (positivo o negativo) dado de forma inmediata en 12 bits sin signo.

    Rd ← (Rn + <offset_12>)  o también  Rd ← (Rn - <offset_12>)

    Ejemplo: LDR R1, [R2, #25] -> R1⬅ (R2 + 25)
- **Indexado** (LDR Rd, [Rn, +/-Rm])
  - Bit 25 = 1 y el desplazamiento es un número dado en un registro indicado en los bits 0-3. La dirección efectiva del dato 2 se obtiene sumándole o restándole a la dirección base (dada por Rn) el dato del registro índice (Rm). En este caso #shift = 0 y SH = 0

    Rd ← (Rn + Rm)  o también  Rd ← (Rn - Rm)

    Ejemplo: LDR R1, [R2, R3] -> R1⬅ (R2 + R3)
- **Indexado con escalado por desplazamiento** (LDR Rd, [Rn, +/-Rm SH #inmediato]
  - Bit 25 = 1 y el desplazamiento es un número dado en un registro indicado en los bits 0-3 al cual se le puede hacer una operación SH previamente. La dirección efectiva del dato 2 se obtiene sumándole o restándole a la dirección base (dada por Rn) el dato operado previamente según lo indicado en SH y #shift . Al igual que en <op2> en la transparencia 26, los desplazamientos pueden ser ASR, LSL, LSR, ROR o RRX

    Rd ← (Rn + Shift(Rm))  o también  Rd ← (Rn – Shift(Rm))

    Ejemplo: LDR R1, [R2, R3 LSL #2] -> R1⬅ (R2 + R3*4)
- **Pre-incremento / Pre-decremento** (LDR Rd, [Rn, #offset]!)
  - Indirecto, pero actualizando el contenido de la dirección base antes hacia arriba (pre /+Rn) o hacia abajo (post/-Rn)

    Ejemplo: LDR R1, [R2, #2] ! -> R2 ⬅ R2 +2 // R1⬅ (R2) -> Este es hacia arriba y como sintácticamente es igual que el indirecto con desplazamiento se pone un ! en la instrucción para diferenciarlos
- **Post-incremento / Post-decremento** (LDR Rd, [Rn], #offset)
  - Indirecto, pero actualizando el contenido de la dirección base después hacia arriba (pre /Rn+) o hacia abajo (post/Rn-)
  - Ejemplo: LDR R1, [R2] #2 -> R1⬅ (R2) // R2 ⬅ R2 +2 -> Este es hacia arriba
- **Importante:** Los bits “P”, “U”, “B”, “W” y “L” están relacionados con estos modos de direccionamiento y también valen para diferenciar ente todas las operaciones LDR y STR de la transparencia 17
  - L -> Es para decir si la instrucciones es LDR (L=1) o STR (L=0)
  - B y W -> Es para diferenciar entre las diferentes instrucciones LDR y STR de la transparencia 17
  - P y U -> Es para indicar si hay incremento o decremento (U) y pre o post (P)

# 5 - Ensamblador vs. Compiladores

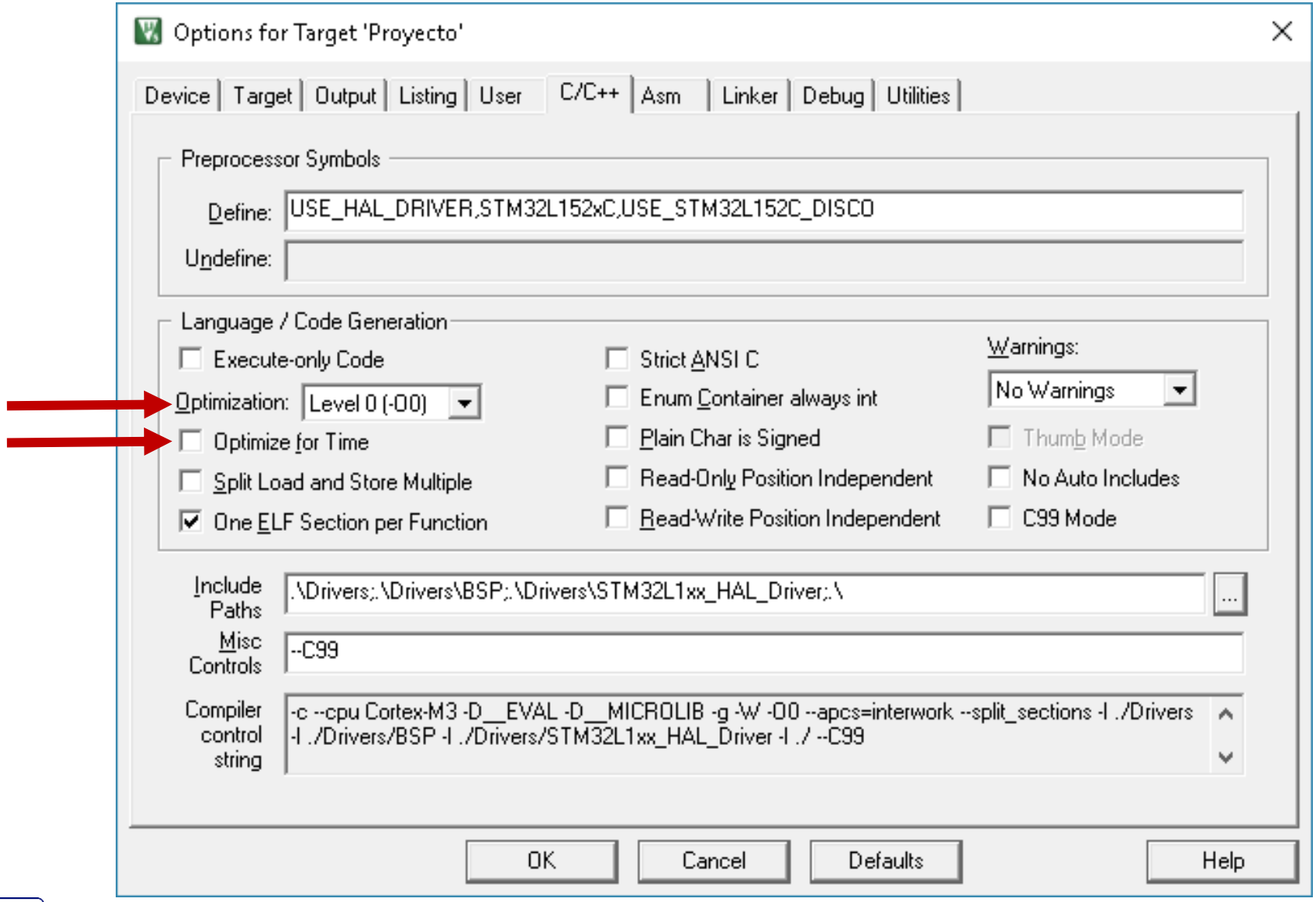
# Ensamblador vs. Compiladores

- Programando en lenguaje ensamblador se puede considerar que cada instrucción corresponde a una instrucción en código máquina
  - Lo que implica en el ARM7 un único ciclo máquina para su ejecución
- Sin embargo, cuando se programa en ensamblador utilizando un entorno "Ensamblador" se tiene que:
  - Se añade al puro lenguaje ensamblador, funcionalidades de medio nivel que incluyen:
    - Definición de constantes
    - Definición de etiquetas, para referirse a otras posiciones del programa:
      - Posición de un salto
      - Ubicación de una variable o una constante
    - Inclusión de macros que faciliten la programación de algunas rutinas complejas y repetitivas
  - Con lo que algunas instrucciones pueden durar varios ciclos máquina
- En cualquier caso, se trabaja muy cercano al microcontrolador, de tal forma que no hay lugar a interpretaciones
  - Como línea general se puede decir que el resultado a nivel de ejecución es independiente del entorno "Ensamblador" utilizado y es la solución más rápida y optimizada posible.

# Ensamblador vs. Compiladores

- Por el contrario, los compiladores de lenguajes de medio y alto nivel tienen que interpretar el código escrito y buscarle una traducción a lenguaje ensamblador
  - Cada compilador puede dar resultados muy distintos, tanto en número de instrucciones utilizadas, como en velocidad de ejecución
- Además tradicionalmente cada compilador suele permitir distintos niveles de optimización, de forma que se busque mejora en:
  - Tiempo de ejecución
  - Tamaño del código
- Esas interpretaciones y traducciones al lenguaje ensamblador ayudan por un lado al programador a desarrollar su solución, al ser mucho más sencilla, pero le hacen perder algo el control de la CPU
  - Además ya no puede saber cuantos ciclos máquina supondrán cada una de sus líneas de código
    - Casi todas las líneas de código escritas, pasan a convertirse en muchas instrucciones de lenguaje ensamblador

# Ensamblador vs. Compiladores



# Ejemplos:

## Opt. Nivel 0 en Código

- En lenguaje C:

```
espera (50000);

void espera(int tiempo) {
   int i;
   for (i=0; i<tiempo; i++);
   }
```

- En ensamblador:

```
0x08002870 MOVW     r0,#0xC350
0x08002874 BL.W     espera
(0x0800285E)

0x0800285E MOVS     r1,#0x00
0x08002860 B        0x08002864
0x08002862 ADDS     r1,r1,#1
0x08002864 CMP      r1,r0
0x08002866 BLT      0x08002862
0x08002868 BX       lr
```

## Opt. Nivel 3 en Código

- En lenguaje C:

```
espera (50000);

void espera(int tiempo) {
   int i;
   for (i=0; i<tiempo; i++);
   }
```

- En ensamblador:

```
0x08001F00 MOVW     r0,#0xC350
0x08001F04 BL.W     espera
(0x08001EEE)

0x08001EF0 B        0x08001EF4
0x08001EF2 ADDS     r1,r1,#1
0x08001EF4 CMP      r1,r0
0x08001EF6 BLT      0x08001EF2
0x08001EF8 BX       lr
```

# Ejemplos:

## Opt. Nivel 0 en Código

- En lenguaje C:

```
espera (50000);

void espera(int tiempo) {
   int i;
   for (i=0; i<tiempo; i++);
   }
```

- En ensamblador:

```
0x08002870 MOVW     r0,#0xC350
0x08002874 BL.W     espera
(0x0800285E)

0x0800285E MOVS     r1,#0x00
0x08002860 B        0x08002864
0x08002862 ADDS     r1,r1,#1
0x08002864 CMP      r1,r0
0x08002866 BLT      0x08002862
0x08002868 BX       lr
```

## Opt. Nivel 3 en Tiempo

- En lenguaje C:

```
espera (50000);

void espera(int tiempo) {
   int i;
   for (i=0; i<tiempo; i++);
   }
```

- En ensamblador:

```
0x080021C8 MOVW     r0,#0xC350
0x080021CC BL.W     espera
(0x080021B6)

0x080021B6 MOVS     r1,#0x00
0x080021B8 CMP      r1,r0
0x080021BA IT       LT
0x080021BC ADDLT    r1,r1,#1
0x080021BE BLT      0x080021B8
0x080021C0 BX       lr
```