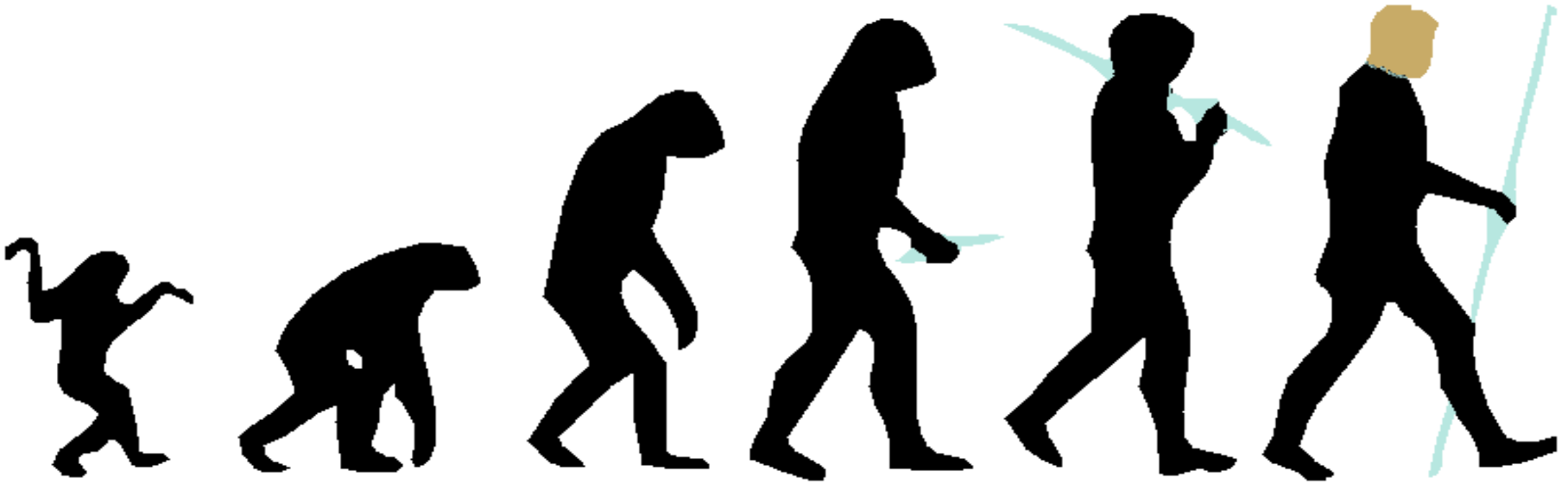
Arquitectura ARM Cortex-M
y
soporte a lenguaje C



2012/07/07

Objetivos

Interpretar un programa ensamblador escrito en Thumb2

Entender la nomenclatura de la documentación sobre ARM Cortex-M

Diferenciar entre Cortex-M3 y Cortex-M4

Saber enumerar las características y componentes básicos de la arquitectura Cortex-M

Diferenciar los pasos del proceso de creación de una aplicación escrita en C

Ser capaz de imaginar el esquema básico de traducción de un fragmento de código C al ensamblador Thumb2

Contenido

1)Conceptos básicos del HW
   1)Unidad de instrucción ARMv7
   2)Segmentación y riesgos

2)Juego de instrucciones Thumb2
   1)Tipos de instrucción
   2)Modificadores

3)Traducción de C a ensamblador
   1)Pasos de compilación

4)Conceptos de C en microcontroladores
   1)Definición de variables y tipos de datos
   2)Depuración y niveles de optimización

ARMv7

La arquitectura ARM nace en 1987 (fecha del primer lanzamiento)

ARMvX : Versión de la arquitectura.

ARMX : Familia de procesadores basados en la misma arquitectura pero con implementaciones diferentes.

ARMv7: última versión de la arquitectura, pero varios diseños diferentes que dan lugar a varias familias:

ARMv7-A : Familia Cortex-A : procesadores de aplicación

ARMv7-M: Familias Cortex-M3 y Cortex-M4: microcontroladores

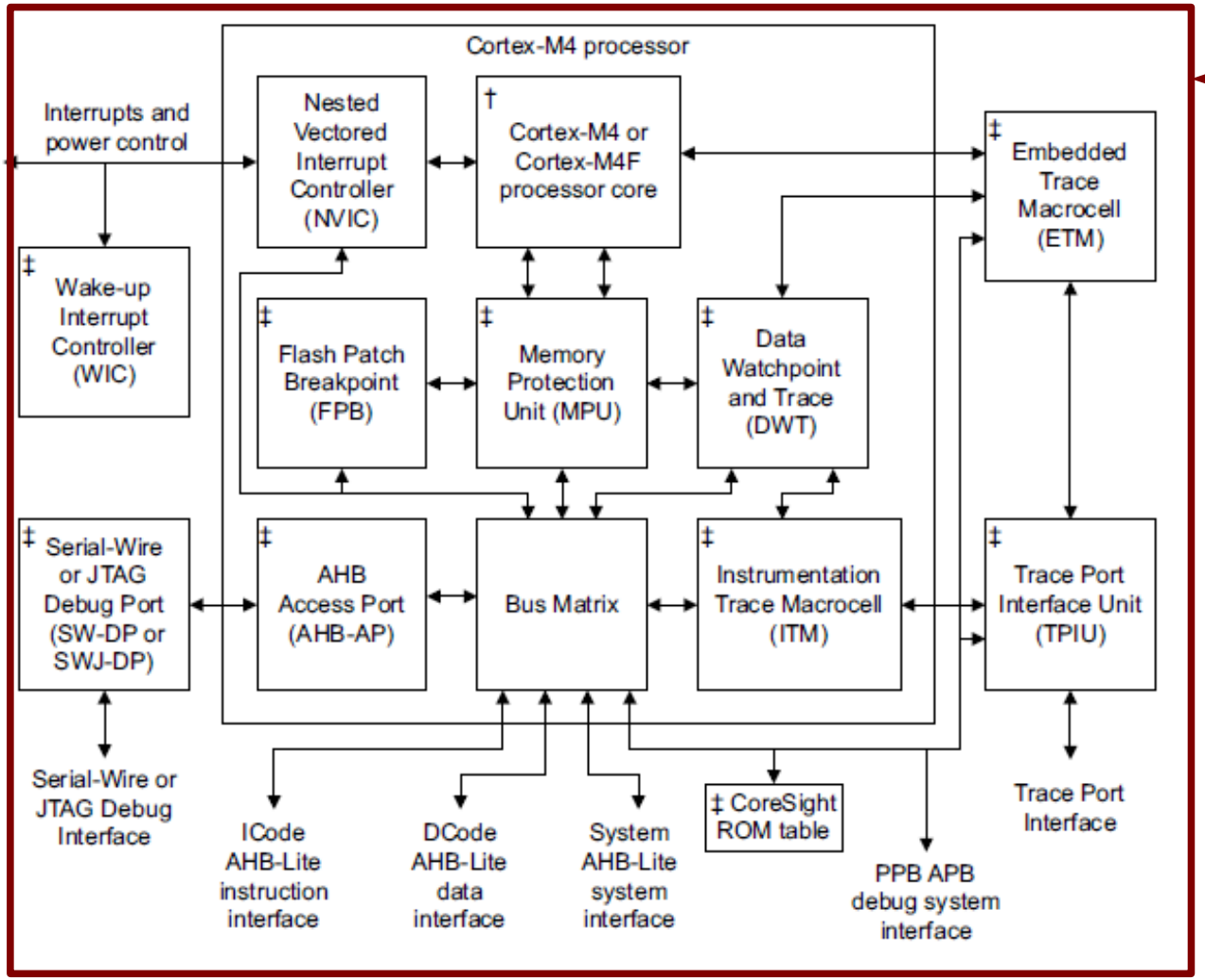
Un diseño ARM define los componentes y su funcionamiento

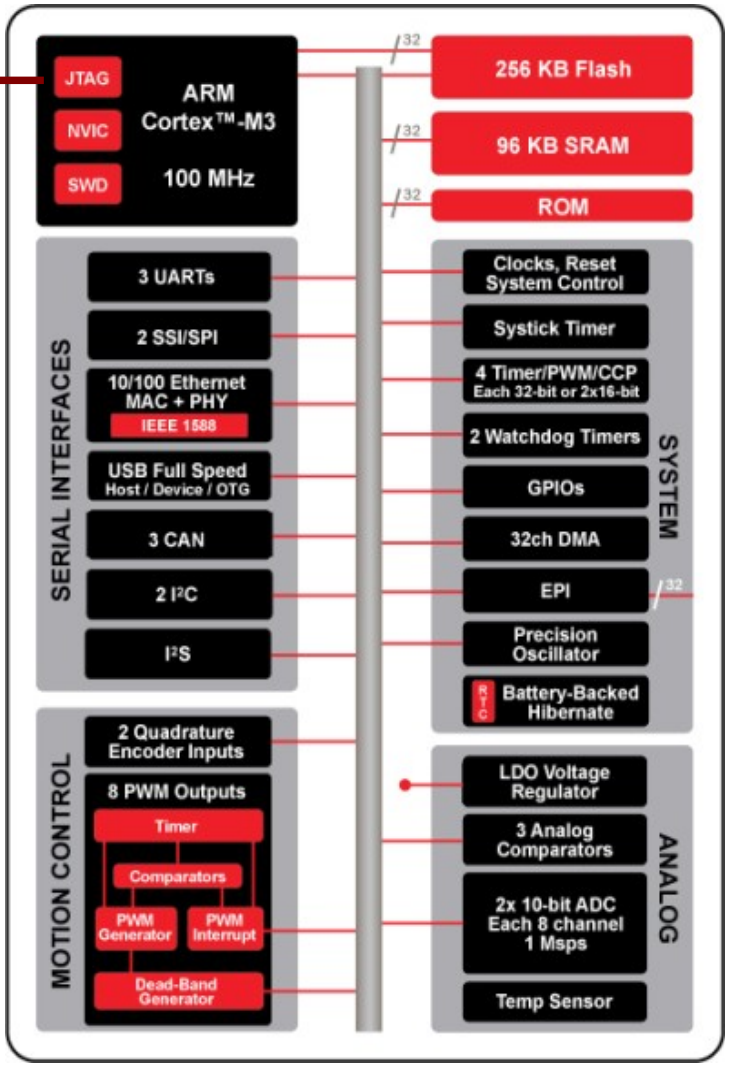
Los diseños se compran... a esto se llama licenciar.

No todos los componentes son obligatorios

Cada licenciado puede hacer modificaciones en la implementación (dentro de las especificaciones)

Esquema de bloques Cortex-M





Texas Instrument

Unidad de Instrucción

- Procesador de 32 bits
- Los registros y las **direcciones de memoria** son de 32 bits
- Juego de instrucciones de tipo LOAD/STORE (REG-REG)
- Unidad de ejecución segmentada en tres etapas.
- Segmentación: se acaba una instrucción por ciclo (Mejor caso)
- Tamaño de las instrucciones: 32 bits (ARM)
- **Thumb**: instrucciones de 16 bots para tener programas compactos
- **Thumb2**: en un mismo código y sin reconfiguración se pueden mezclar instrucciones de 16 y de 32 bits
- Arquitectura HARVARD
- Saltos con especulación (Aplicable a saltos condicionales)
  - Temporización exacta de bucles más complicada → Imposible

ARMv7-M

¿Por qué Thumb 2?

Código ARM: todas las instrucciones, con todas las opciones y todos los modos de direccionamiento

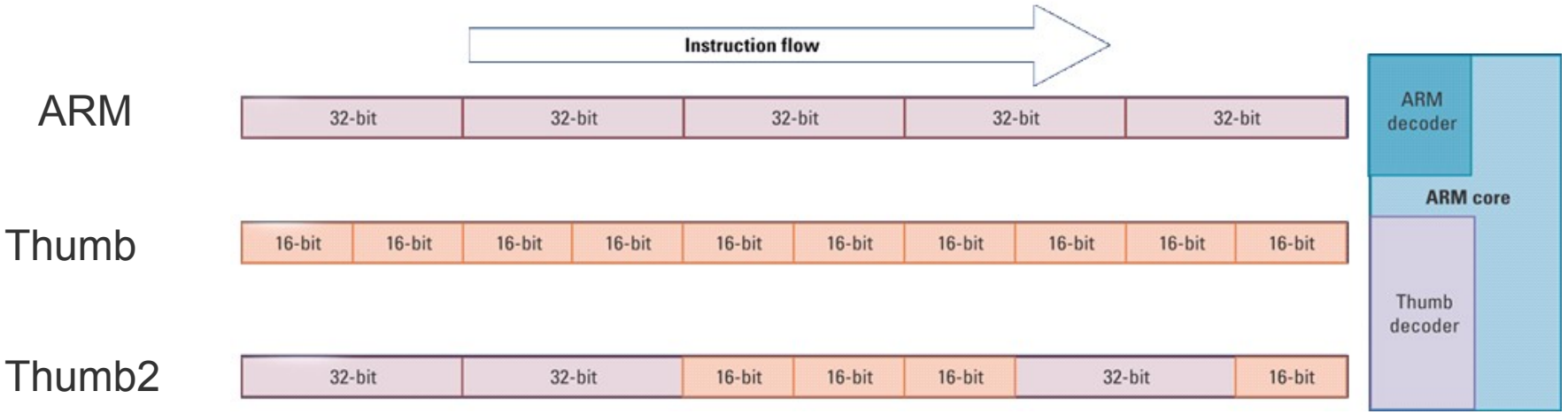
Código Thumb: instrucciones de 16 bits más “compactas”.

Objetivo: programas más cortos

Problema: NO cabe lo mismo que en 32 bits, programas más lentos

Thumb2: combina los dos anteriores de forma que no sea necesario poner instrucciones para pasar de un modo a otro.

No es necesario que las instrucciones estén alineadas a palabra.



Registros accesible por las instrucciones

16 registros de 32 bits... pero:

- R15 ==> PC
- R13 ==> SP
- R14 ==> LR

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Table 2, Core registers and AAPCS usage

Ejecución segmentada

Segmentación: 1 instrucción por ciclo salvo **RIESGOS**

- RIESGO == perdida de prestaciones == STALL
- **Stall**: significa que hay un ciclo en el que no podemos empezar una nueva instrucción porque hay otra que lo impide

Cada fabricante indica con una tabla las “latencias” de las instrucciones que pueden generar “stalls”

- **Latencia**: tiempo que se tarda en que una instrucción “acabe”

Cortex-M: evita muchos riesgos por HW

- No todos los riesgos se pueden evitar
- Riegos de datos y de nombre: cortocircuitos + compilador
- Riesgos de de control: especulación (apostar qué va a ocurrir)
- VER: Cortex-M3 Technical Referencce Manual → Programmers Model → Instruction set summary

Memoria

Espacio de direccionamiento “plano” con direcciones de 32 bits

MPU (opcional): Memory Protection Unit

- Permite definir cómo se accede a la memoria (hasta 8 zonas)

Dispositivos: mapeados en memoria

- No todo el espacio de direccionamiento se puede usar para RAM

Arquitectura HARVARD: el procesador accede a memoria por dos caminos diferentes, uno para datos y otro para programa.

ENDIAN: puede ser big o little

Zonas especiales para acceso eficiente a bits: BIT BANDS

Mapa de Memoria

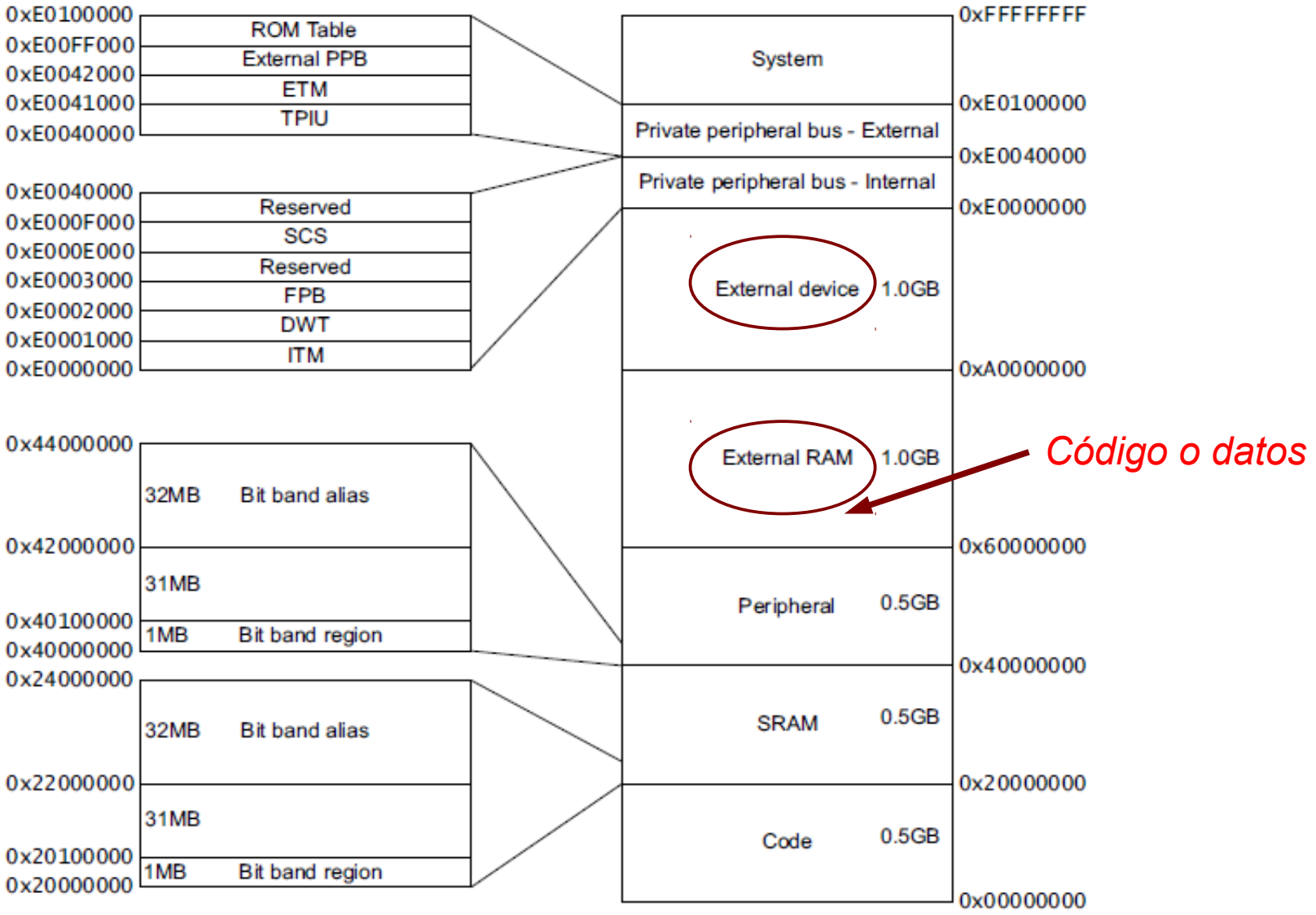


Figure 3-1 System address map

Mapa de Memoria

Dispositivos:

De 0x4000000 a 0x60000000 definidos por el fabricante

De 0xA000000 a 0xE0000000 definidos por el fabricante

De 0xE000000 a 0xE0040000 definidos por ARM

De 0xE004000 a 0xE0100000 definidos por ARM o el fabricante

De 0xE010000 a 0xFFFFFFFF definidos por el fabricante

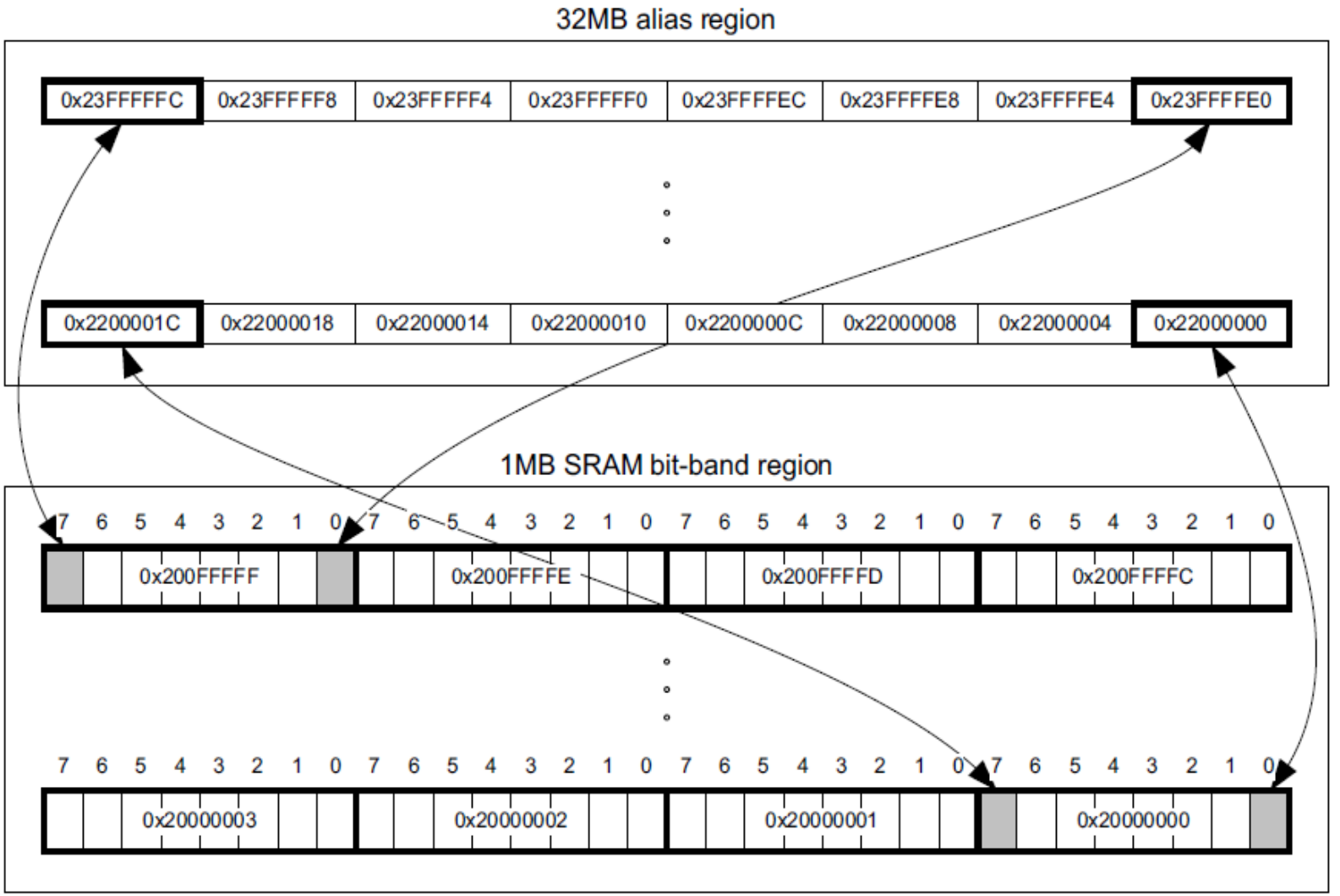
Memoria:

De 0x0000000 a 0x20000000 2 vías de acceso (Dcode y Icode)

De 0x2000000 a 0x40000000 1 vía de acceso (System Bus)

De 0x6000000 a 0xA0000000 1 vía de acceso (System Bus)

Acceso al Bit Band



dir_pal_bit= (dir_base_bit_band)+(dir_palabra*32)+(n_bit*4)

Caminos de interconexión

CortexTM -M3 core is designed for System on a Chip (SOC) approaches.

The System-Level bus interfaces from the CortexTM -M3 core to microcontroller manufacturer are based on the AHB and APB protocols.

Advanced Microcontroller Bus Architecture (AMBA) describes these protocols.

Advanced High-Performance Bus (AHB)

- Instruction bus (I-code bus): 32 bit
- Data bus (D-code bus): 32 bit
- System Bus: 32 bit

(Internal) Advanced Peripheral Bus (APB): 32 bit

- External Private Peripheral Bus (External PPB): 32 bit

Caminos de interconexión

Caminos de interconexión STM32F4

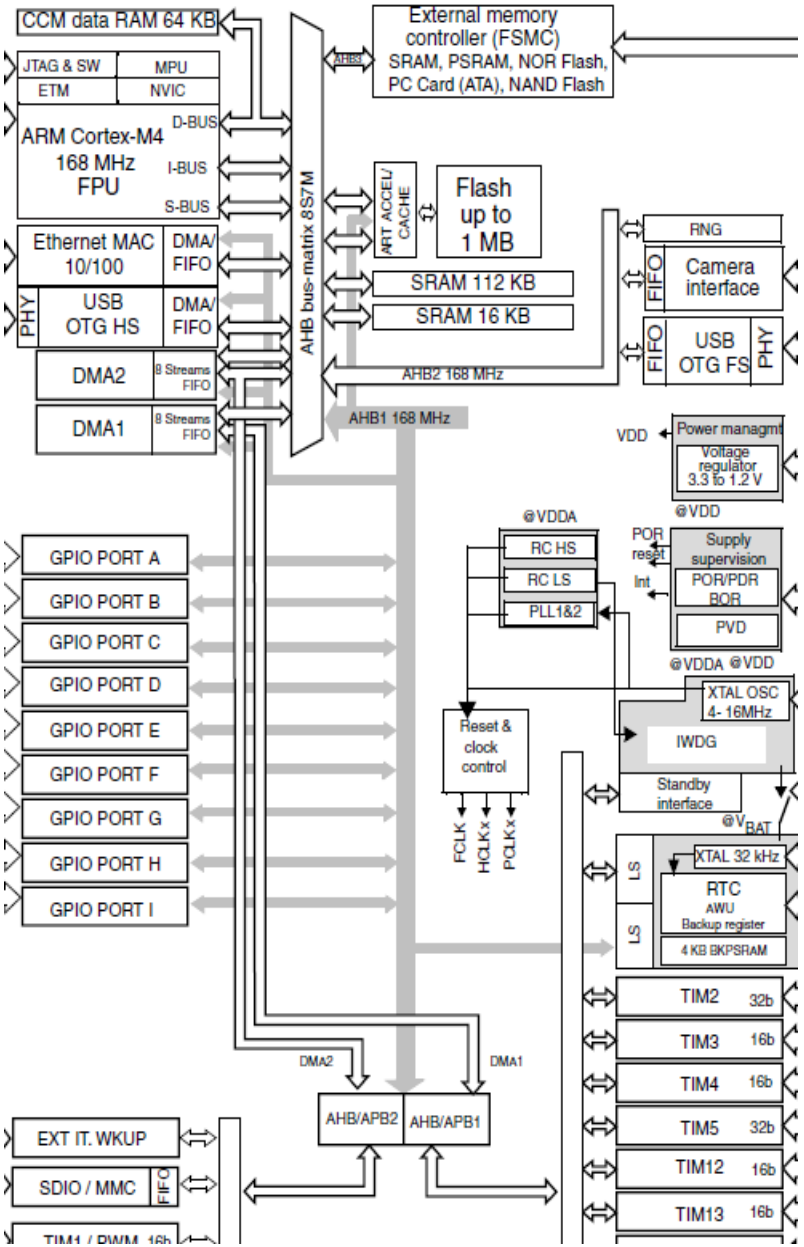
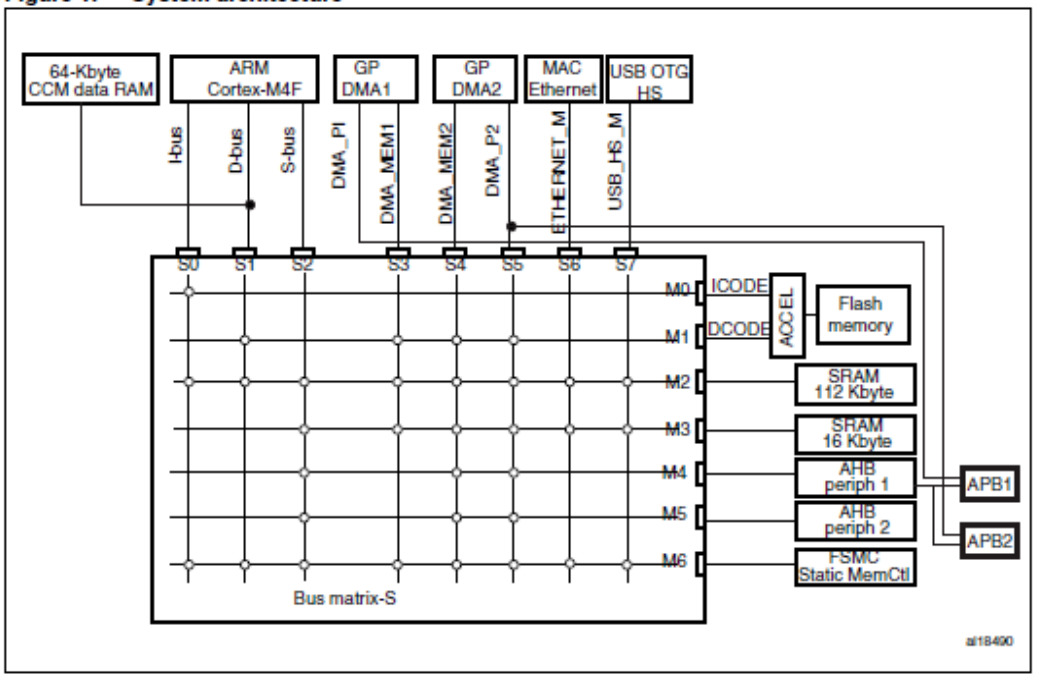


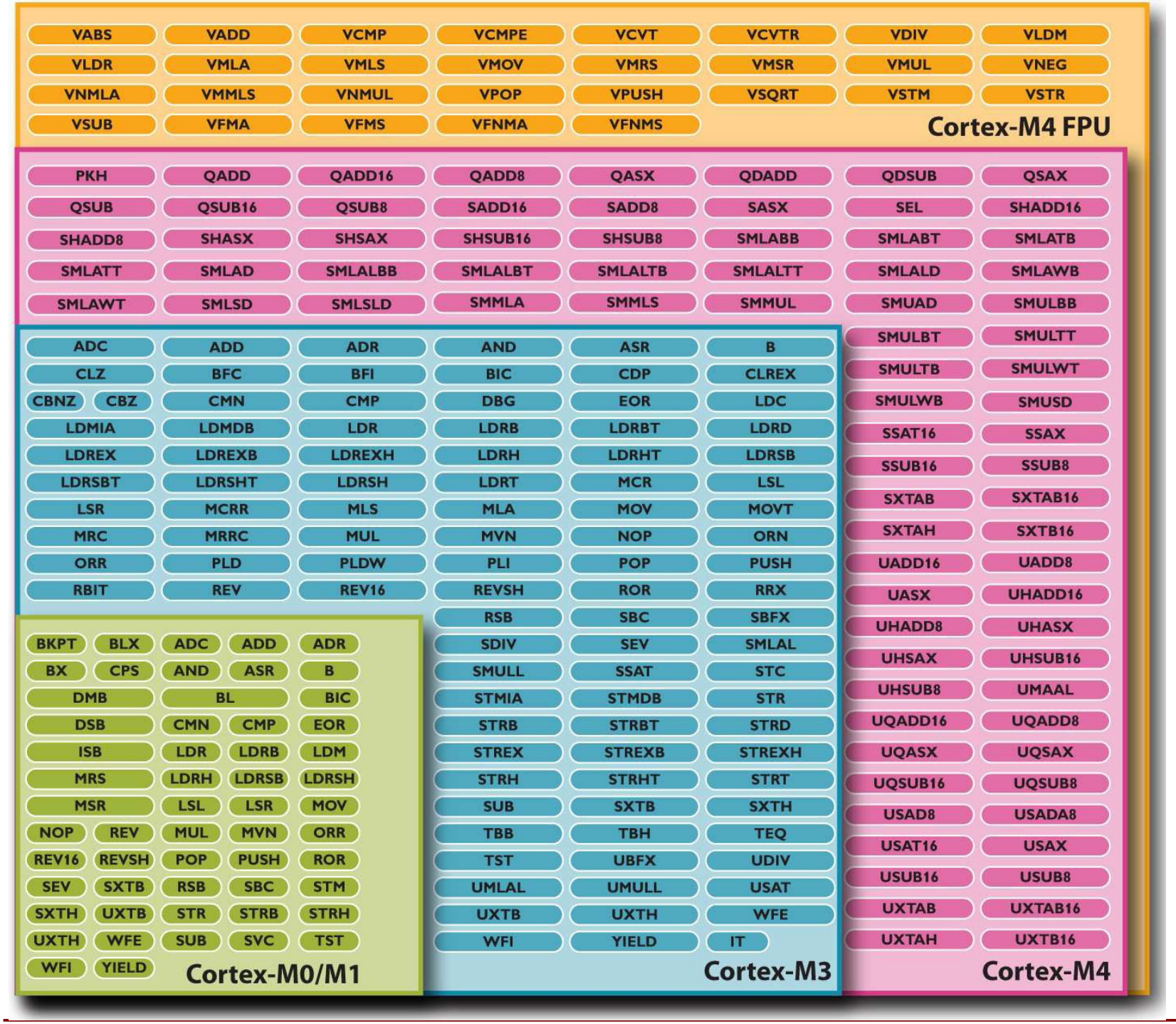
Figure 1. System architecture



ISA: Tipos básicos de Instrucción

http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001m/QRC0001_UAL.pdf

- Aritmético-lógicas
  - Fuentes en registros o valores inmediatos
  - Destino siempre un registro
- Movimiento entre registros (MOV)
  - Fuentes y destino en registros o valores inmediatos
- Acceso a memoria
    - Sólo con instrucciones LOAD y STORE
- Control:
  - Salto (B, BL, BX, BLX, BXJ, CBZ, CBNZ, TBB, TBH)
      - ¡CUIDADO! El PC es un registro a todos los efectos
  - Instrucción IT (If-Then) y Ejecución condicional
  - Modificar el estado del procesador


VABS VADD VCMP VCMPE VCVT VCVTR VDIV VLDM
VLDR VMLA VMLS VMOV VMRS VMSR VMUL VNEG
VNMLA VMMLS VNMUL VPOP VPUSH VSQRT VSTM VSTR
VSUB VFMA VFMS VFNMA VFNMS
Cortex-M4 FPU
PKH QADD QADD16 QADD8 QASX QDADD QDSUB QSAX
QSUB QSUB16 QSUB8 SADD16 SADD8 SASX SEL SHADD16
SHADD8 SHASX SHSAX SHSUB16 SHSUB8 SMLABB SMLABT SMLATB
SMLATT SMLAD SMLALBB SMLALBT SMLALTB SMLALTT SMLALD SMLAWB
SMLAWT SMLSD SMLSLD SMMLA SMMLS SMMUL SMUAD SMULBB
ADC ADD ADR AND ASR B SMULBT SMULTT
CLZ BFC BFI BIC CDP CLREX SMULTB SMULWT
CBNZ CBZ CMN CMP DBG EOR LDC SMULWB SMUSD
LDMIA LDMDB LDR LDRB LDRBT LDRD SSAT16 SSAX
LDREX LDREXB LDREXH LDRH LDRHT LDRSB SSUB16 SSUB8
LDRSBT LDRSHT LDRSH LDRT MCR LSL SXTAB SXTAB16
LSR MCRR MLS MLA MOV MOVT SXTAH SXTB16
MRC MRRC MUL MVN NOP ORN UADD16 UADD8
ORR PLD PLDW PLI POP PUSH UASX UHADD16
RBIT REV REV16 REVSH ROR RRX UHADD8 UHASX
RSB SBC SBFX UHSAX UHSUB16
SDIV SEV SMLAL UHSUB8 UMAAL
SMULL SSAT STC UQADD16 UQADD8
STMIA STMDB STR UQASX UQSAX
STRB STRBT STRD UQSUB16 UQSUB8
STREX STREXB STREXH USAD8 USADA8
STRH STRHT STRT USAT16 USAX
SUB SXTB SXTH USUB16 USUB8
TBB TBH TEQ UXTAB UXTAB16
TST UBFX UDIV UXTAH UXTB16
UMLAL UMULL USAT
UXTB UXTH WFE
WFI YIELD IT
Cortex-M4
Cortex-M3
BKPT BLX ADC ADD ADR
BX CPS AND ASR B
DMB BL BIC
DSB CMN CMP EOR
ISB LDR LDRB LDM
MRS LDRH LDRSB LDRSH
MSR LSL LSR MOV
NOP REV MUL MVN ORR
REV16 REVSH POP PUSH ROR
SEV SXTB RSB SBC STM
SXTH UXTB STR STRB STRH
UXTH WFE SUB SVC TST
WFI YIELD
Cortex-M0/M1

Operaciones Aritmético/lógicas

- Ensamblador basado en “Quick Reference Card”:

  ADD{S}<c> {Rd,}Rn,<Operand2>

- Las llaves indican opcional, luego esto describe al menos dos operaciones: ADD y ADDS
  - Con la S se actualizan flags : **N**egativo, **Z**ero, **C**arry, o**V**erflow.
- **<c>** Ejecución condicional basada en flags actuales.
- El registro destino es Rd, si no está, Rn es fuente y destino
- <Operand2> puede ser:
  - Valor inmediato
  - Registro
  - Registro sobre el que se hace una operación de desplazamiento

Operando flexible con desplazamiento

Rm, shift

- Shift puede ser:
  - ASR #n Desplaza. Derecha aritmético (1<=n<=32)
  - LSR #n Desplaza. Derecha lógico (1<=n<=32)
  - LSL #n Desplaza. Derecha lógico (1<=n<=31)
  - ROR #n Rotación a la derecha (1<=n<=31)
  - RRX Rotación a la derecha de un bit que implica al bit de acarreo
- Permite otros tipos de especificación menos utilizados. Podéis encontrar su especificación en el manual de la arquitectura ARMv7 si os registráis en ARM como usuarios
- Pueden haber restricciones en el uso de algunos registros en algunas operaciones (sobre todo PC y SP)

Movimiento entre registros

MOV{s}<c> Rd, <Op2> Copia <Op2> en Rd

MVN{s}<c> Rd, <Op2> Copia <Op2> en Rd negado bit a bit

MOVT<c> Rd,#<i16> Copia el valor <i16> en la parte alta de Rd

MOV<c> Rd,#<i16> Copia el valor <i16> en la parte baja de Rd

- CUIDADO: una operación sobre PC o R15 equivalen a un salto
- CUIDADO: la operaciones sobre R13 o SP modifican la pila
- La actualización de los códigos de condición dependen del valor escrito en el registro destino.

Operaciones LOAD/STORE

Dos tipos: sencillas y múltiples

- Todas: utilizan un registro base para calcular la dirección de memoria
  - El registro bases se puede modificar antes o después de calcular la dirección de acceso a memoria
    - Antes: LDR R3, [R5, #14]!
    - Después: LDR R3, [R5], #14
- Sencillas: utilizan un registro para almacenar resultado (LOAD) o leer el dato (STORE)
  - Pueden usarse para leer bytes, medias palabra o palabras
- Multiples: copian o leen múltiples registros de posiciones consecutivas de memoria
  - Todos los valores de registros en memoria serán de 32 bits

Instrucciones de Control

B<c> <label> : salto a <label> se anula dependiende de <c>

BL<c> <label> : equiv a B<c> <label> pero guarda PC en LR

BX<c> Rm : pone en valor de RM en PC

BLX : No util en Thumb-2 equivalente a BL o a BX

CB{N}Z Rn, label: Salta a <label> si RN es cero (Z) o no (NZ)

TBB TBH : Salto a través de tabla de direcciones (switch)

IT{patern} <c>: Condiciona las siguientes instrucciones según <c>

IT, ITT, ITE, ITTT, ITTE, ITET, ITEE, ITTTT, ITTET, ITETT, ITEET, ITTTE, ITTEE, ITETE, ITEEE

Más “barato” que un salto... culpa de la segmentación

NO todas las instrucciones se pueden usar en bloque condicional

En modo ARM no es necesario, útil en THUMB

Ejecución Condicional

ITE NE
SUBNE R0,R1,R2
ADDEQ R0,R4,R3

SUBEQ R1,R2,R3

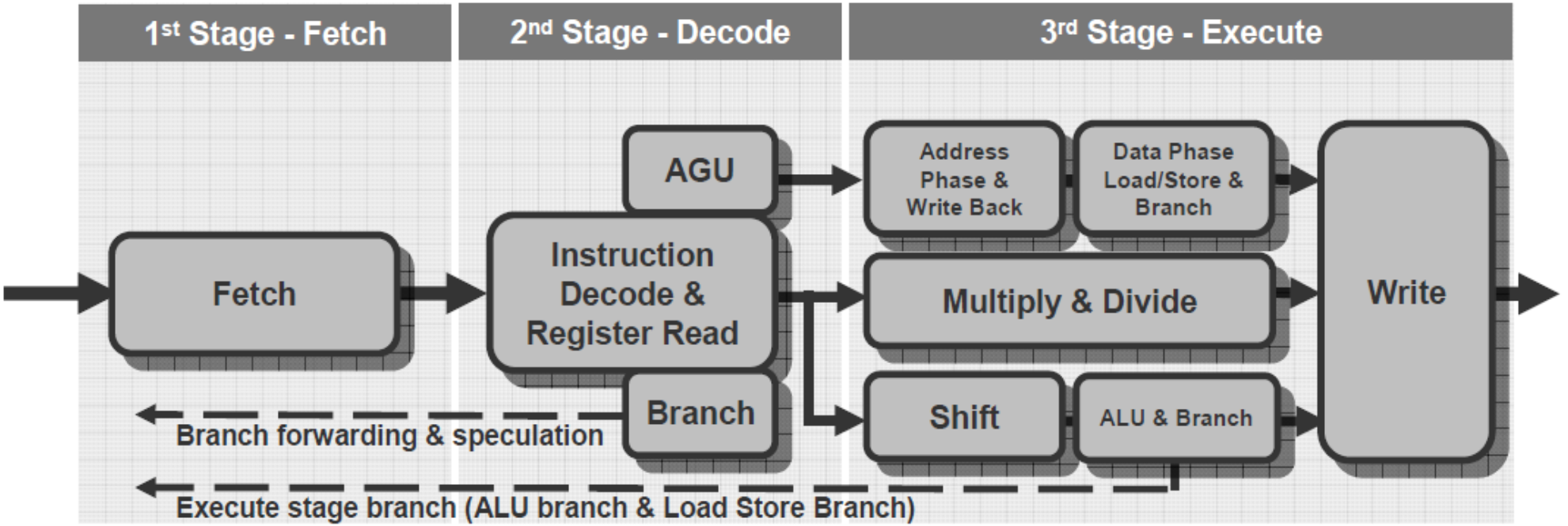
SUBNE R1,R3,R2

Table A7-1 Condition codes

cond	Mnemonic extension	Meaning, integer arithmetic	Meaning, floating-point arithmetic[a]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal, or unordered	Z == 0
0010	CS[b]	Carry set	Greater than, equal, or unordered	C == 1
0011	CC[c]	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Not unordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C == 1 and Z == 0
1001	LS	Unsigned lower or same	Less than or equal	C == 0 or Z == 1
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 and N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
1110	None (AL)[d]	Always (unconditional)	Always (unconditional)	Any

a. Unordered means at least one NaN operand.
b. HS (unsigned higher or same) is a synonym for CS.
c. LO (unsigned lower) is a synonym for CC.
d. AL is an optional mnemonic extension for always, except in IT instructions. See *IT* on page A7-277 for details.

Unidad Segmentada



¿FUNCIONA IGUAL QUE SI NO ESTUVIERA SEGMENTADO?

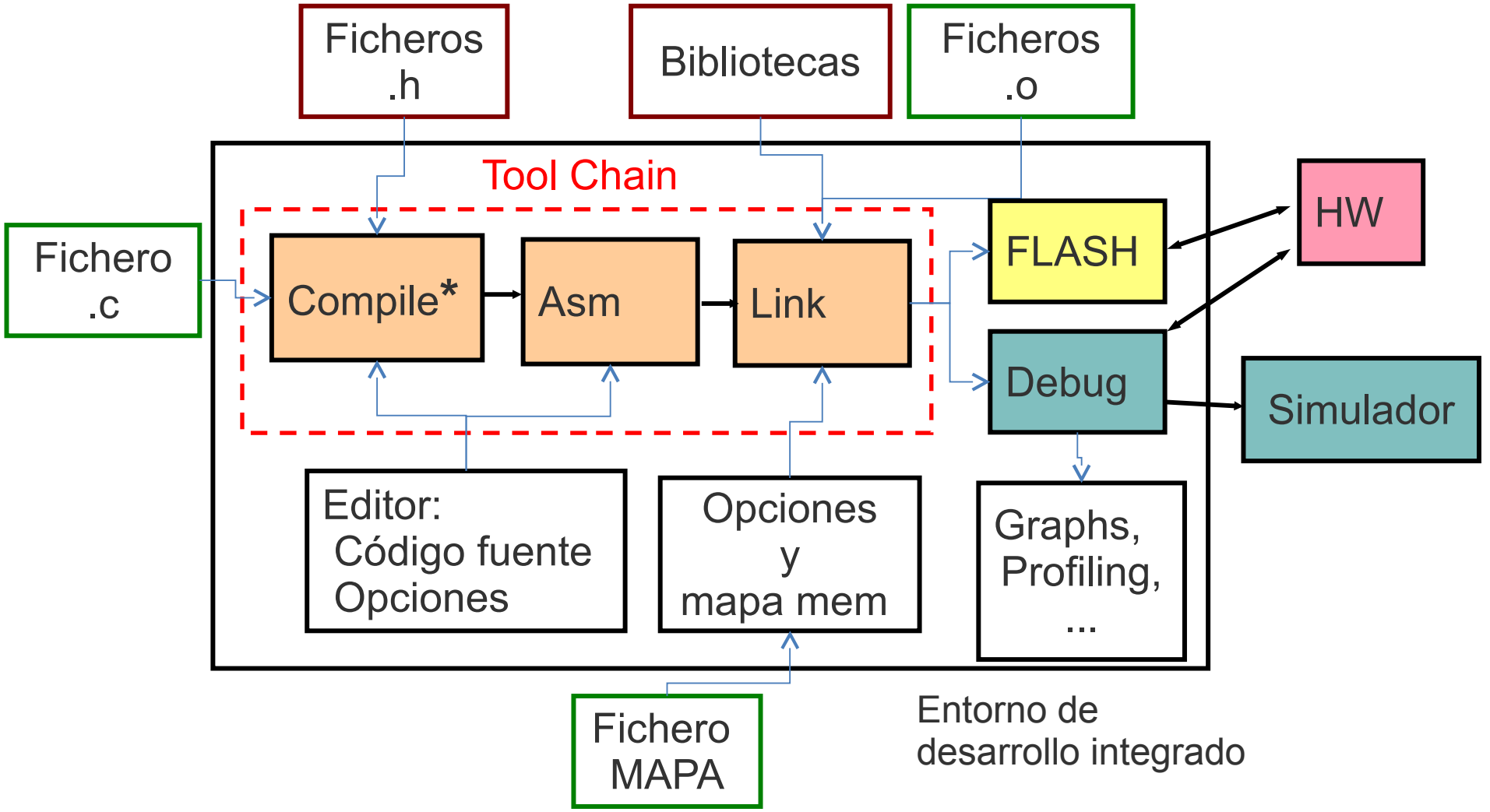
¿QUÉ PASA CON LOS LOAD/STORE MÚLTIPLES?

¿QUÉ PASA CON LOS SALTOS?

¿QUÉ PASA CON LAS DIVISIONES?

En blanco intencionadamente

Proceso de creación de aplicaciones



Compilación

Acciones:

- Comprueba que el código de entrada no tiene errores
- Traduce a ensamblador: **NO CREA APLICACIONES**

Componentes:

- Precompilador:
  - Elimina todas las líneas que empiezan por # menos #pragma
  - #include: Busca ficheros .h y los “concatena” junto al fichero .c
    - Los .h **NO SON LIBRERIAS**
  - Elimina todos los comentarios (esta acción se puede evitar)
- Compilador: traducción de C a ensamblador
  - La traducción depende de la máquina destino el compilador usado
  - Diferentes tipos de traducción: normal, optimizada para espacio, optimizada para velocidad...

Ensamblador

- Codifica el ensamblador (todavía era un archivo de texto) para que lo pueda leer la CPU destino
- SALIDA: archivo objeto o binario
  - Son las piezas de la futura aplicación: *SEPARADAS*
  - La salida del ensamblador no se puede leer directamente
    - Cada 'toolchain' incluye herramientas específicas para su manipulación
  - Los ficheros objeto se pueden organizar en ficheros más grandes llamados bibliotecas o librerías
  - Archivos objetos de dos toolchain diferentes pueden no ser compatibles entre sí, aunque sean para el mismo procesador

Enlazador

- Une los diferentes componentes de un programa según un patrón
  - El patrón lo determina la toolchain empleada, la máquina donde se ejecutará y el sistema operativo utilizado.
  - Cuando se trabaja con MCU no **suele** haber S.Op.
- Determina las posiciones de memoria de variables y funciones
  - ***BINDING***
  - Este proceso se basa en los nombres generados por el compilador a partir del código fuente
  - Si hay una referencia a un símbolo y no se encuentra la implementación (función) o la definición (variable) de ese símbolo se genera un error y la aplicación no se crea
  - Si un símbolo no se referencia el enlazador **suele** eliminarlo.
- Utiliza un "MAPA" para decidir donde pone cada elemento

Programación en C

- Tipos básicos
  - Enteros sin especificación exacta de tamaño:
    - {unsigned|signed} int
    - {unsigned|signed} char
    - {unsigned|signed} short {int}
    - {unsigned|signed} long {int}
    - {unsigned|signed} long long {int}
  - Coma flotante
    - float (32 bits en ARM)
    - double (64 bits en ARM)
  - Punteros
    - Su tamaño depende de la máquina: 32 bits en ARM

Tipos C99: stdint.h

- Define tipos con tamaño en bits definido e independiente de la máquina o compilador:
  - int**N**_t y uint**N**_t : entero de N bits con signo y sin signo
  - int_fast**N**_t: indica aquel entero de al menos N bits que es el más rápido en la implementación actual
  - int_least**N**_t: indica el entero con la codificación más pequeña de al menos N bits en la implementación actual
  - **intptr_t**: entero capaz de almacenar una dirección de memoria
  - intmax_t: entero de la longitud más grande que se puede manejar en la implementación actual
- Tamaños especificados para N: 8 ,16, 32 y 64

Programación en C

- Tipos compuestos: se componen de elementos de cualquier tipo válido de C (básicos y compuestos)
  - Vectores
    - Conjunto de elementos del mismo tipo que se almacenan consecutivos en memoria y se acceden con indice.
  - Registros
    - Conjunto de elementos de tipos diversos.
    - Se acceden asignando un identificador a cada componente que estará en una posición de memoria diferente
    - ALINEACIÓN: como se separán los diferentes campos
    - ***CAMPOS DE BITS***
  - Enumeration: es una forma alternativa de crear constantes
  - Union: parece a un registro, pero todos los componentes se alinean para usar la misma zona de memoria

CAMPOS DE BITS

- Ejemplo:

```
struct MI_BYTE{
    uint8_t b1:1;
    uint8_t b2:1;
    uint8_t res:5;
    uint8_t l1:1;
}
```

- Ejemplo2:

```
struct MI_DATO2{
    uint8_t b1:1;
    uint8_t b2:1;
    uint8_t res:5;
    uint32_t b:10;
    uint32_t c:15;
}
```

__attribute__((packed))

Programación en C

Funciones

- Trozos de código autocontenidos y que ofrecen encapsulación de algoritmos y datos
- Pueden tener cero o más parámetros y devolver hasta un único valor
- Se pueden definir variables para almacenar funciones...
  - Lo que se almacena es la dirección de inicio del código
- CUIDADO
  - Las variables dentro de una función sólo existen **mientras** se ejecuta la función, luego su memoria se libera: no devolver nunca un puntero a una de estas variables → Están en la PILA
  - Las parámetros en C siempre se pasan por valor: una copia de la variable
    - Se puede simular paso por referencia con punteros

La memoria y C

Un programa en C supone dividida la memoria en secciones. Los tipos de secciones que contempla son:

- Código: zona de memoria donde está el programa en ejecución.
- Datos globales:
  - Zona de memoria donde están las variables accesibles desde cualquier función o aquellas que se deben conservar entre llamadas a la misma función (**static**)
  - Zona de memoria donde el compilador pone constantes que no puede poner como valores inmediatos
- **PILA**:
  - Zona dinámica donde se crean variables de las funciones
  - Zona auxiliar para guardar el estado entre llamadas a funciones o ante la llegada de interrupciones
- **Heap**: zona dinámica gestionada por el programador

¿Dónde está mi variable?

- Si se ha definido fuera de una función se le asigna una posición de memoria fija
- Si se ha definido dentro de una función:
  - Puede estar en la pila
  - Puede que el compilador haya decidido mapearla a un registro (más frecuente si se ha optimizado el código?
- CUIDADO
  - Como los LOAD/STORE son caros, el compilador trata de minimizar estas acciones al optimizar, por lo que incluso las variables definidas fuera de cualquier función pueden estar mapeadas temporalmente a un registro.
    - Esta acción depende del compilador: no siempre ocurre

Modificadores estándar

- const: el contenido de esta variable es de SÓLO lectura
- extern: el símbolo al que se refiere existe... ya lo verá el linker
- register: indica que esta variable se **puede** poner en un registro de la CPU sin temor a efectos colaterales.
- static: este símbolo sólo se puede acceder desde aquí
  - Fuera de función: acceso sólo desde este fichero
  - Dentro de función sobre variable: la variable NO está en la pila y mantiene el valor que tenía la última vez que se ejecutó la función
- **volatile**: indica que esta variable debe "estar en memoria y sólo en memoria"
  - El compilador no la mapea a ningún registro
  - En CMSIS se definen sinónimos como **__IO** , **__I** y **__O**

Condiciones en C

- No existe un tipo específico para almacenar datos BOOL
  - Se utiliza el valor entero “cero” para indicar FALSO y “no cero” para cierto.
  - Las comparaciones devuelven enteros
  - Cualquier operación se puede usar como “condición”
- Estructuras condicionales simples:
  - if(cond){cuerpo_cierto}
  - if(cond){cuerpo_cierto}else{cuerpo_falso}
- Operadores condicionales de comparación: <,>,<=,>=,==,!=
- Operadores lógicos: !,&&,||
- Operador condición (no es un if): result=(cond)?valor1:valor2;
- Las condiciones compuestas SIEMPRE con “early exit”

Bucles

- Tres tipos de bucles en C:
  - for(init;cond;inc) { cuerpo }
  - while(cond){cuerpo}
  - do{cuerpo}while(cond);
- Se pueden realizar modificaciones para implementar cualquiera de ellos con otro.
- En sistemas segmentados el do{}while() suele ser el esquema más eficiente:
  - Ejecuta menos instrucciones de saltos si se dan al menos dos pasadas por el bucle.
  - Aunque uses otro tipo de bucle el compilador lo pasa a esta estructura... no te preocupes.

Entorno de Depuración

- Depuración y optimización son una mala combinación si no se conoce MUY BIEN el ensamblador de la máquina
- Sin optimización el programa puede ser más lento o más grande (o ambos).
  - Hay diferentes niveles de optimización
  - Cada compilador realiza unas acciones diferentes en cada nivel
- CONSEJO:
  - No actives la optimización en la fase de diseño de un programa a menos que no te quepa en memoria o sea demasiado lento.
  - QUITA la generación de código e información de depuración para la aplicación final: más pequeña, más rápida y más difícil de "hackear"

MANOS A LA OBRA

Seguir los pasos del profesor en pizarra y pantalla

PREGUNTAD

PREGUNTAD

PREGUNTAD