ARM

Advanced RISC Machines

Conjunto de Instrucciones ARM

Características principales del Conjunto de Instrucciones ARM

- Todas las instrucciones son de 32 bits.
- La mayoría de las instrucciones se ejecutan en un ciclo.
- Cada instrucción se puede ejecutar condicionalmente.
- Una arquitectura load/store
  - Instrucciones de procesamiento de datos actúan solo en registros
    - Formato de tres operandos
    - ALU y shifter combinados para una manipulación rápida de bits
  - Instrucciones de acceso a memoria específicas con potentes modos indexados de direccionamiento
    - 32 bit y 8 bit tipos de datos
      - Y también 16 bit tipos de datos en ARM 4.
    - Instrucciones de registro multiple flexible load and store
- Conjunto de extensión de instrucciones via coprocesadores

Modos del Procesador

- ARM tiene seis modos de operación:
  - *User* (unprivileged mode under which most tasks run)
  - *FIQ* (entered when a high priority (fast) interrupt is raised)
  - *IRQ* (entered when a low priority (normal) interrupt is raised)
  - *Supervisor* (entered on reset and when a Software Interrupt instruction is executed)
  - *Abort* (used to handle memory access violations)
  - *Undef* (used to handle undefined instructions)
- Arquitectura ARM Versión 4 agrega un séptimo modo:
  - *System* (privileged mode using the same registers as user mode)

Los Registros

- ARM tiene 37 registros en total, todos son de 32-bits.
  - 1 dedicado al contador de programa PC
  - 1 dedicado al registro de estado en curso
  - 5 dedicados a salvar el registro de estado de programa
  - 30 registros de propósito general
- Sin embargo, estos están dispuestos en varios bancos, con el banco accessible comandado por el modo del procesador. Cada modo puede accesar
  - Un conjunto particular r0-r12 registros
  - Uno particular r13 (the stack pointer) y r14 (link register)
  - r15 (contador de programa)
  - cpsr (registro de estado del programa corriendo)

  y modos priviligeados tienen acceso a
  - Un particular spsr (registro de estado salvado del programa)

Organización de Registros

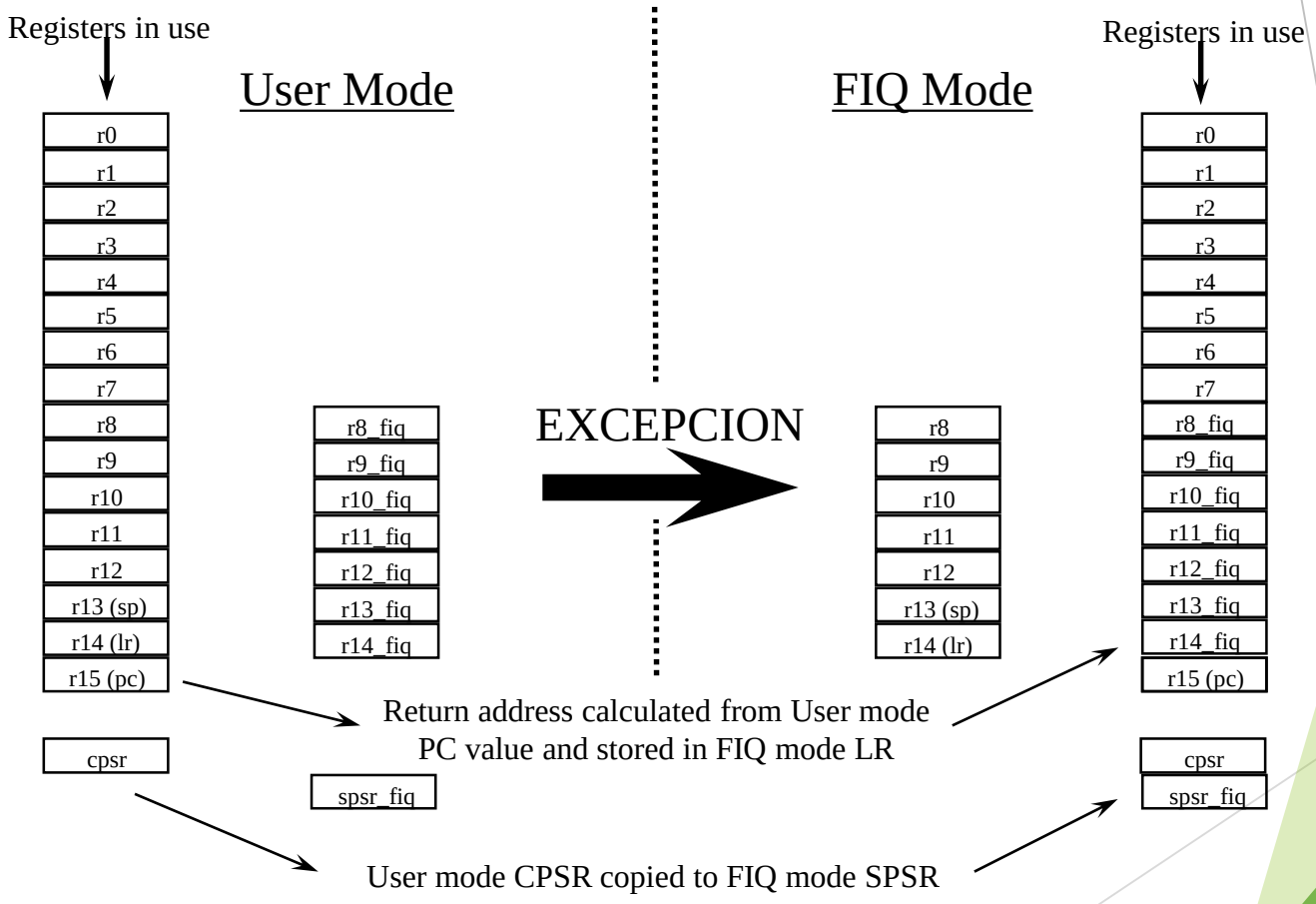
Registros generales y Contador de Programa

User32 / System	FIQ32	Supervisor32	Abort32	IRQ32	Undefined32
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	r8_fiq	r8	r8	r8	r8
r9	r9_fiq	r9	r9	r9	r9
r10	r10_fiq	r10	r10	r10	r10
r11	r11_fiq	r11	r11	r11	r11
r12	r12_fiq	r12	r12	r12	r12
r13 (sp)	r13_fiq	r13_svc	r13_abt	r13_irq	r13_undef
r14 (lr)	r14_fiq	r14_svc	r14_abt	r14_irq	r14_undef
r15 (pc)	r15 (pc)	r15 (pc)	r15 (pc)	r15 (pc)	r15 (pc)

Program Status Registers

cpsr	cpsr	cpsr	cpsr	cpsr	cpsr
	spsr_fiq	spsr_svc	spsr_abt	spsr_irq	spsr_undef

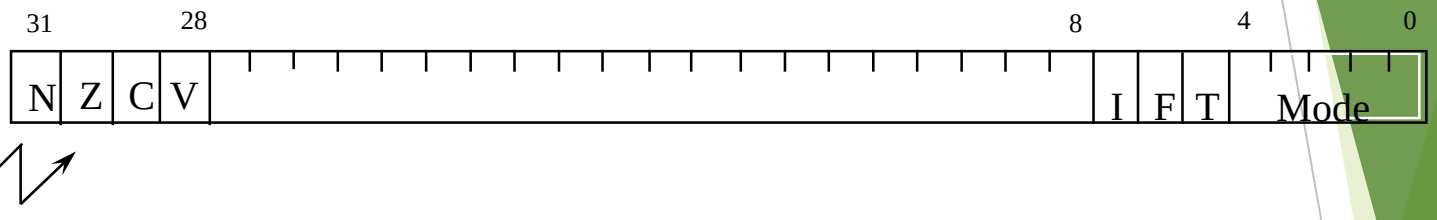
Ejemplo de registro: Usuario a modo FIQ



Accesando Registros usando Instrucciones ARM

- Registros accesibles.
  - Todas las instrucciones pueden accesar r0-r14 directamente.
  - La mayoría de las instrucciones permiten el uso del PC.
- Instrucciones específicas para accesar CPSR y SPSR.
- Nota : En modo priviligeado, es también possible cargar / almacenar el modo usuario de registros hacia o desde la memoria.
  - Pronto detalles de esto.

El registro de Estado del Programa (CPSR and SPSRs)



Copias de los flags de estados de la ALU
(latched si las intrucciones tienen el bit "S" set).

* Flags de códigos de condición

  N = Resultado Negativo desde el ALU flag.
  Z = Resultado Zero desde el ALU flag.
  C = Operación ALU genera Carry
  V = Operación ALU Overflowed

* Mode Bits

  M[4:0] define el modo del procesador.

* Bits de desabilitación de Interrupción.

  I = 1, desabilita la IRQ.
  F = 1, desabilita el FIQ.

* T Bit (Arquitectura v4T)

  T = 0, Procesador en estado ARM
  T = 1, Procesador en estado Thumb

Flags de Condición

Flag	Instrucciones Lógicas	Instrucciones Aritméticas
Negativo (N=‘1’)	Sin significado	Bit 31 del resultado se colocó en set Indica un número negativo en operaciones con signo
Zero (Z=‘1’)	Resultado es todo ceros	Resultado de la operación fue cero
Carry (C=‘1’)	Después de la operación Shift se dejó un ‘1’ en el carry flag	Resultado fue mayor que 32 bits
Overflow (V=‘1’)	Sin significado	Resultado fue mayor que 31 bits Indica una posible corrupción en el bit de signo de los números

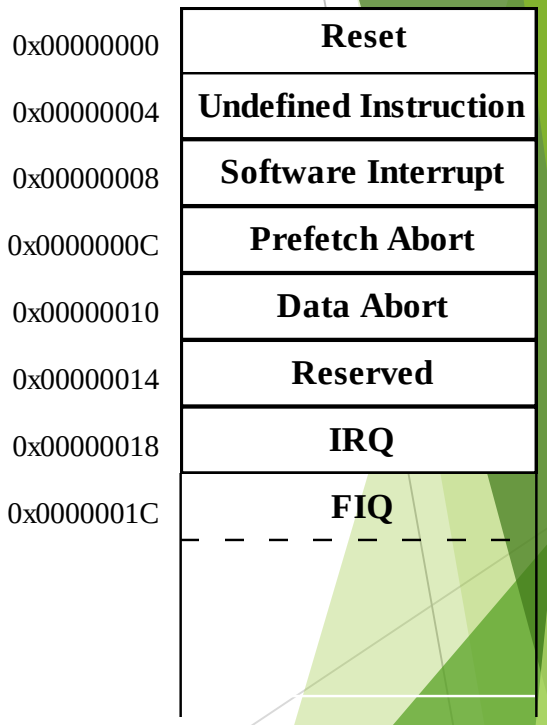
El Contador de Programa (R15)

- Cuando el procesador está ejecutando en el estado ARM:
  - Todas las instrucciones son de 32 bits
  - Todas las instrucciones deben estar alineadas por palabra
  - Entonces el valor del PC está almacenado en bits [31:2] con bits [1:0] iguales a cero(la instrucción no puede ser halfword o alineada por byte).
- R14 se usa como el registro link de subrutina (LR) y almacena la dirección de retorno cuando operaciones Branch con Link se ejecutan calculadas desde el PC.
- Así para retornar desde un linked Branch
  - `MOV r15,r14`

  o
  - `MOV pc,lr`

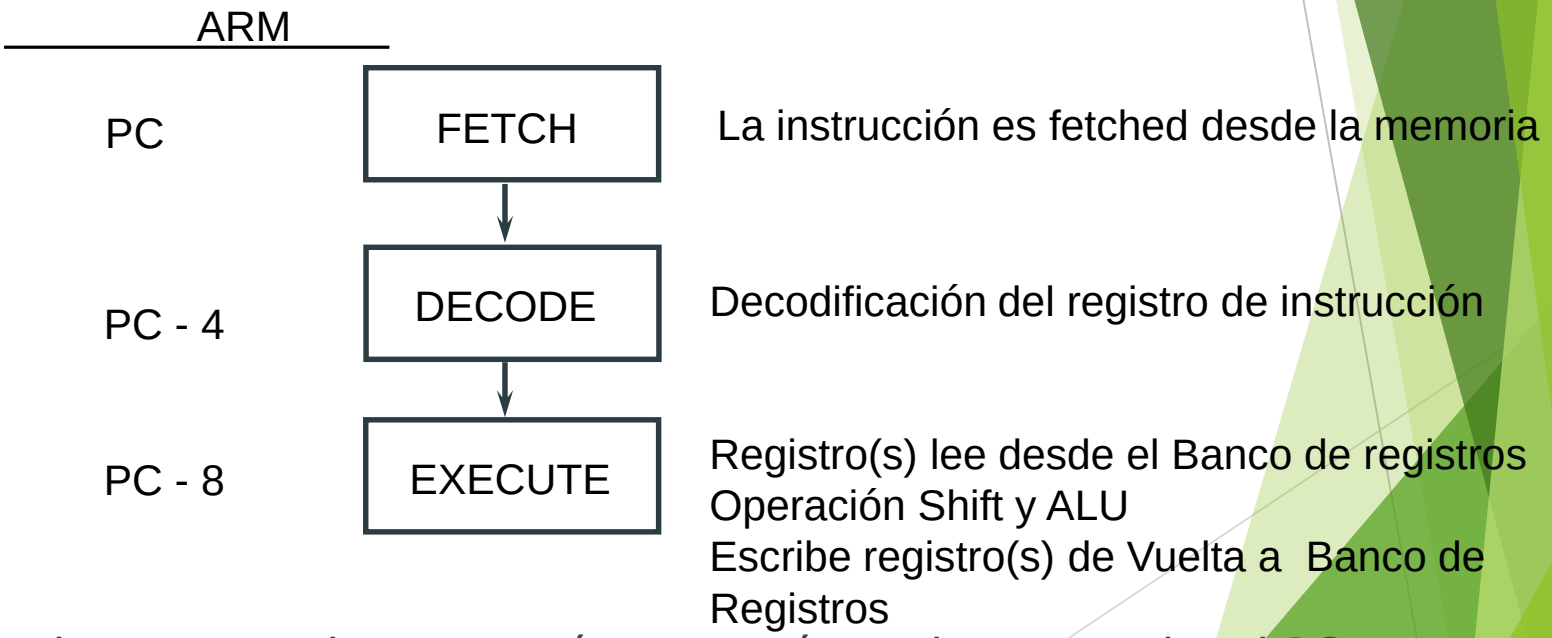
Manejo de excepciones y el vector de Tabla

- Cuando ocurre una excepción, el núcleo:
  - Copia CPSR en SPSR_<mode>
  - Coloca bits apropiados en CPSR
    - Si el core implementa instrucciones ARM arquitectura 4T y está en estado Thumb , entonces
      - Se entra al estado ARM .
    - Bits de campo del modo
    - La interrupción desabilita flags si es pertinente.
  - Mapeo in registros apropiados
  - Almacena la "*dirección de retorno*" en LR_<mode>
  - Coloca el PC con el vector de dirección
- Para retornar, el manejo de excepción necesita:
  - Restaurar CPSR desde SPSR_<mode>
  - Restaurar PC desde LR_<mode>



El pipeline de Instrucción

- ARM usa un pipeline para aumentar la velocidad del flujo de instrucciones en el procesador
  - Permite que varias operaciones se ejecuten simultáneamente, en vez de serialmente



- En vez de apuntar a la instrucción que está siendo ejecutada, el PC apunta a la instrucción que está siendo traída desde memoria (fetched).

Formato del Conjunto de Instrucciones ARM

31 2827 1615 87 0

Cond	0 0	I	Opcode				S	Rn	Rd	Operand2					
Cond	0 0 0 0 0 0					A	S	Rd	Rn	Rs	1 0 0 1			Rm	
Cond	0 0 0 0 1			U	A	S		RdHi	RdLo	Rs	1 0 0 1			Rm	
Cond	0 0 0 1 0			B	0 0			Rn	Rd	0 0 0 0	1 0 0 1			Rm	
Cond	0 1	I	P	U	B	W	L	Rn	Rd	Offset					
Cond	1 0 0		P	U	S	W	L	Rn	Register List						
Cond	0 0 0		P	U	1	W	L	Rn	Rd	Offset1	1	S	H	1	Offset2
Cond	0 0 0		P	U	0	W	L	Rn	Rd	0 0 0 0	1	S	H	1	Rm
Cond	1 0 1	L	Offset												
Cond	0 0 0 1		0 0 1 0					1 1 1 1	1 1 1 1	1 1 1 1	0 0 0 1			Rn	
Cond	1 1 0	P	U	N	W	L		Rn	CRd	CPNum	Offset				
Cond	1 1 1 0		Op1					CRn	CRd	CPNum	Op2		0	CRm	
Cond	1 1 1 0		Op1			L		CRn	Rd	CPNum	Op2		1	CRm	
Cond	1 1 1 1		SWI Number												

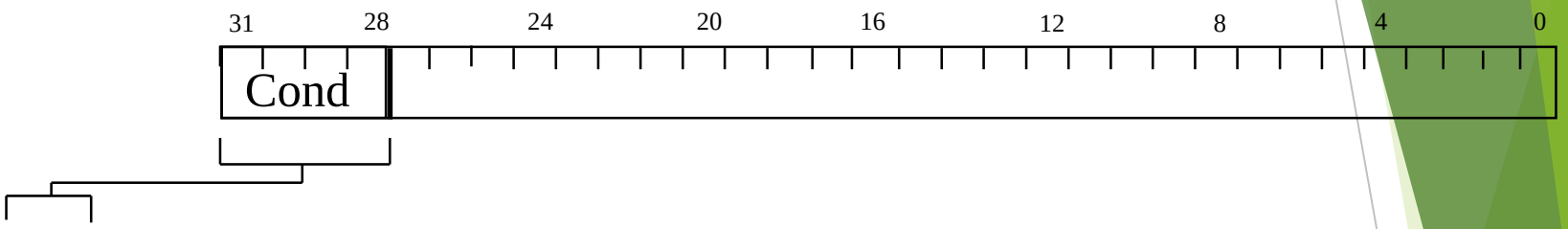
Tipo de Instrucción

- Data processing / PSR Transfer
- Multiply
- Long Multiply (v3M / v4 only)
- Swap
- Load/Store Byte/Word
- Load/Store Multiple
- Halfword transfer : Immediate offset (v4 only)
- Halfword transfer: Register offset (v4 only)
- Branch
- Branch Exchange (v4T only)
- Coprocessor data transfer
- Coprocessor data operation
- Coprocessor register transfer
- Software interrupt

Ejecución Condicional

- Muchos conjuntos de instrucciones solo permiten que los Branches se ejecuten condicionalmente
- Sin embargo, reusando el hardware de evaluación de condición, ARM efectivamente aumenta el número de instrucciones.
  - Todas las instrucciones tienen un campo de condición el cual determina si la CPU lo ejecuta
  - Las instrucciones no ejecutadas consumen 1 cycle.
    - Igual se tiene que completer un ciclo para permitir el fetching y decoding de las siguientes instrucciones.
- Esto elimina la necesidad de muchos Branches, lo que detiene el pipeline (3 cycles para volver a completar).
  - Permite código in-line muy denso, sin branches.
  - La pérdida de tiempo por no ejecutar varias instrucciones condicionales es frecuentemente menor que el overhead del branch o el llamado a subrutina que podría necesitarse.

El Campo de Condición



0000 = EQ - Z set (equal)

0001 = NE - Z clear (not equal)

0010 = HS / CS - C set (unsigned higher or same)

0011 = LO / CC - C clear (unsigned lower)

0100 = MI -N set (negative)

0101 = PL - N clear (positive or zero)

0110 = VS - V set (overflow)

0111 = VC - V clear (no overflow)

1000 = HI - C set and Z clear (unsigned higher)

1001 = LS - C clear or Z (set unsigned lower or same)

1010 = GE - N set and V set, or N clear and V clear (>or =)

1011 = LT - N set and V clear, or N clear and V set (>)

1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)

1101 = LE - Z set, or N set and V clear,or N clear and V set (<, or =)

1110 = AL - always

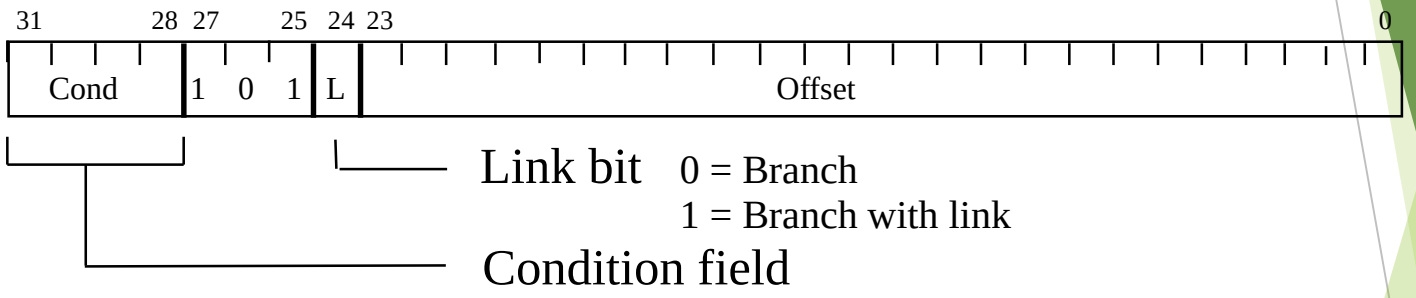
1111 = NV - reserved.

Usando y Renovando el Campo de Condición

- Para ejecutar una instrucción condicionalmente, simplemente se debe colocar con la condición apropiada
  - Por ejemplo, una instrucción add toma la forma:
    - `ADD r0,r1,r2  ; r0 = r1 + r2 (ADDAL)`
  - Se ejecuta solo si el flag Zero está en 1(set):
    - `ADDEQ r0,r1,r2  ; If zero flag set then…`
      `; ... r0 = r1 + r2`
- Normalmente, operaciones de procesamiento de datos no afectan los flags de condición(aparte de las comparaciones dónde esto es el único efecto). Para renovar los flags de condición, el bit S de la instrucción debe estar en 1 en la instrucción(y cualquier código de condición) con una "S".
  - Por ejemplo, para sumar dos números y colocar los flags de condición:
    - `ADDS r0,r1,r2  ; r0 = r1 + r2  ; ... and set flags`

Instrucciones Branch(1)

- Branch : `B{<cond>} label`
- Branch con Link : `BL{<cond>} sub_routine_label`



- El offset para instrucciones branch lo calcula el assembler:
  - Tomando la diferencia entre la instrucción branch y la dirección destino menos 8 (para permitir el pipeline).
  - Esto da un offset de 26 bits el cual es desplazado a la derecha 2 bits (como los dos últimos son siempre cero porque las instrucciones estan alineadas por palabras) y almacenados en la codificación de la instrucción.
  - Esto da un rango de ± 32 Mbytes.

Instrucciones Branch(2)

- When executing the instruction, the processor:
  - shifts the offset left two bits, sign extends it to 32 bits, and adds it to PC.
- Execution then continues from the new PC, once the pipeline has been refilled.
- The "Branch with link" instruction implements a subroutine call by writing PC-4 into the LR of the current bank.
  - i.e. the address of the next instruction following the branch with link (allowing for the pipeline).
- To return from subroutine, simply need to restore the PC from the LR:
  - `MOV pc, lr`
  - Again, pipeline has to refill before execution continues.
- The "Branch" instruction does not affect LR.
- Note: Architecture 4T offers a further ARM branch instruction, BX
  - See Thumb Instruction Set Module for details.

Instrucciones de Procesamiento de Datos

- La familia más larga de instrucciones ARM, todas comparten el mismo formato de instrucciones.
- Contiene:
  - Operaciones aritméticas
  - Comparaciones
  - Operaciones lógicas
  - Movimiento de datos entre registros
- Recuerde, esto es una arquitectura load / store
  - Estas instrucciones solo trabajan en registros, ***NO*** memoria.
- Cada una de ellas realiza una operación específica en uno o dos operandos
  - El primer operando es siempre un registro- Rn
  - Segundo operando se envia a la ALU via el barrel shifter.
- Examinaremos el Barrel shifter.

Operaciones Aritméticas

- Estas operaciones son:
  - ADD operand1 + operand2
  - ADC operand1 + operand2 + carry
  - SUB operand1 - operand2
  - SBC operand1 - operand2 + carry -1
  - RSB operand2 - operand1
  - RSC operand2 - operand1 + carry - 1
- Sintaxis:
  - <Operation>{<cond>}{S} Rd, Rn, Operand2
- Ejemplos:
  - ADD r0, r1, r2
  - SUBGT r3, r3, #1
  - RSBLES r4, r5, #5

Comparaciones

- El único efecto de una comparación es
  - ***UPDATE THE CONDITION FLAGS***. No es necesario colocar(set) el bit S.
- Las operaciones son:
  - CMP operand1 - operand2, resultado no se escribe
  - CMN operand1 + operand2, resultado no se escribe
  - TST operand1 AND operand2, resultado no se escribe
  - TEQ operand1 EOR operand2, resultado no se escribe
- Sintaxis:
  - <Operation>{<cond>} Rn, Operand2
- Ejemplos:
  - CMP r0, r1
  - TSTEQ r2, #5

Operaciones lógicas

- Las operaciones son:
  - AND operand1 AND operand2
  - EOR operand1 EOR operand2
  - ORR operand1 OR operand2
  - BIC operand1 AND NOT operand2 [ie bit clear]
- Sintaxis:
  - <Operation>{<cond>}{S} Rd, Rn, Operand2
- Ejemplos:
  - AND r0, r1, r2
  - BICEQ r2, r3, #7
  - EORS r1,r3,r0

Movimiento de Datos

- Las operaciones son:
  - MOV operand2
  - MVN NOT operand2

  Notar que estas no hacen uso del operand1.
- Sintaxis:
  - <Operation>{<cond>}{S} Rd, Operand2
- Ejemplos:
  - MOV r0, r1
  - MOVS r2, #10
  - MVNEQ r1,#0

El Barrel Shifter

- ARM no tiene instrucciones de desplazamiento (shift).
- Tiene un barrel shifter el cual proporciona un mecanismo para hacer desplzamientos como parte de otras instrucciones
- Así, qué operaciones soporta el barrel shifter?

Barrel Shifter – Shift a la izquierda

- Shifts izquierda por una cantidad específica(multiplica por potencia de 2) e.g.

  LSL #5 = multiplica por 32

Logical Shift Left (LSL)

Barrel Shifter – Shift a la derecha

Shift Lógico a la Derecha

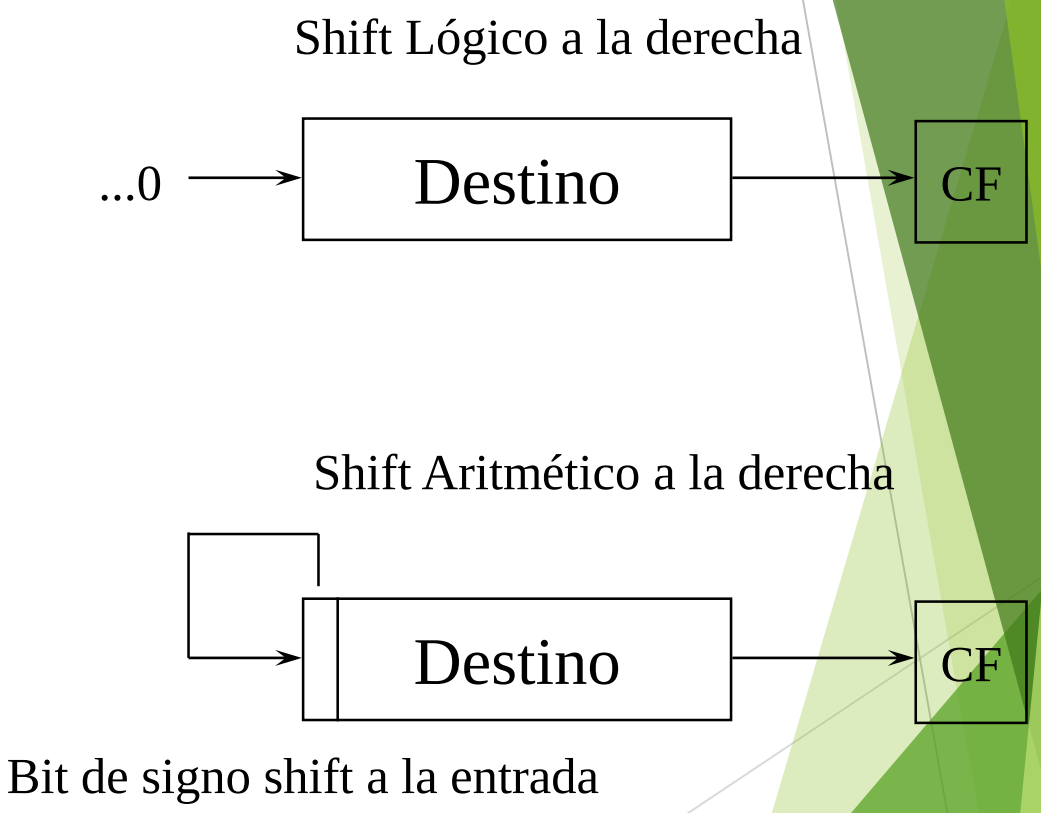
- Shifts a la derecha por una cantidad específica(divide por potencia de 2) e.g.

  LSR #5 = divide por 32

Shift Aritmético a la Derecha

- Shifts a la derecha (divide por potencia de 2) y preserva el bit de signo, para operaciones en complemento 2. e.g.

  ASR #5 = divide por 32



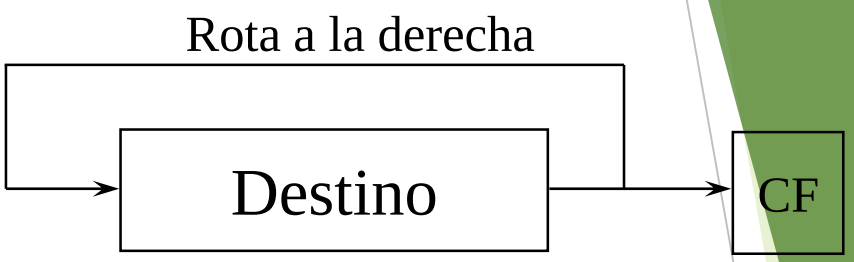
Barrel Shifter - Rotaciones

Rotación a la Derecha(ROR)

- Similar a un ASR pero los bits wrap around cuando dejan el LSB y aparecen como el MSB.

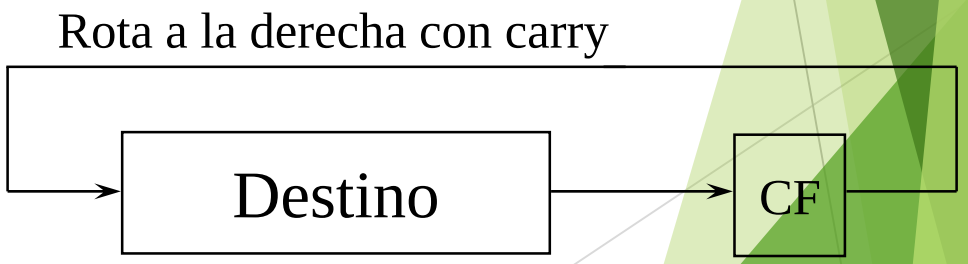
  e.g. ROR #5

- Nota: El último bit rotado se usa como Carry Out.

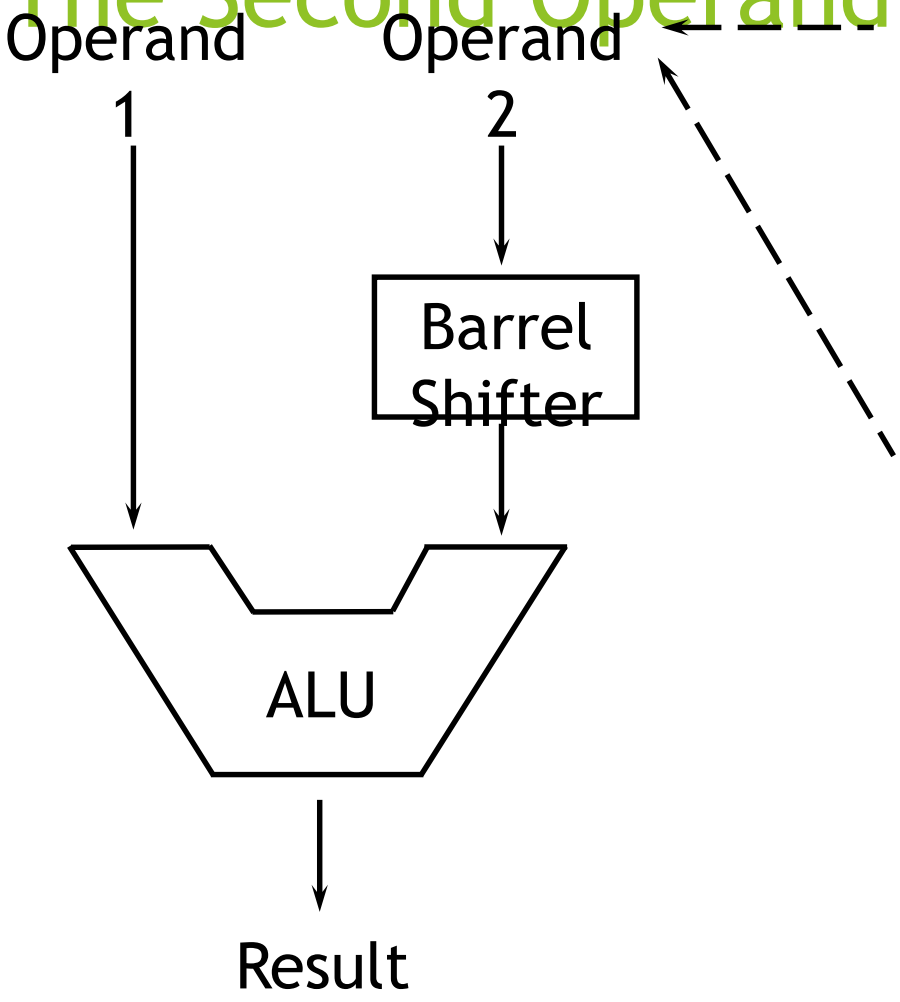


Rotación a la Derecha Extendida(RRX)

- Esta operación usa el flag CPSR C como un 33rd bit.
- Rota a la derecha por 1 bit. Codificado como ROR #0.



Using the Barrel Shifter: The Second Operand



- Register, optionally with shift operation applied.
- Shift value can be either be:
  - 5 bit unsigned integer
  - Specified in bottom byte of another register.

* Immediate value

- 8 bit number
- Can be rotated right through an even number of positions.
- Assembler will calculate rotate for you from constant.

Segundo Operando: Registro desplazado

- La cantidad por el cual el registro es desplazado esta contenido en una de éstas:
  - El campo inmediato de 5-bit en la instrucción
    - ***NO OVERHEAD***
    - Desplazamiento es realizado sin costo- ejecuta en un ciclo.
  - El ultimo byte de un registro (no el PC)
    - Entonces toma un ciclo extra para ejecutar
    - ARM no puede leer 3 registros de una vez.
    - Entonces, igual como en otros procesadores dónde el shift es una instrucción separada.
- Si no se específica un shift, entonces se aplica uno por default: LSL #0
  - i.e. el barrel shifter no tiene efecto sobre el valor del registro.

Segundo Operando: Usando un Registro Desplazado

- El usar una instrucción para multiplicar por una constant significa primero cargar la constant en un registro y después esperar un número de ciclos para que se complete la instrucción.
- Una mejor solución es usando una combinación de MOVs, ADDs, SUBs y RSBs con shifts.
  - Multiplicaciones por una constante igual a ((power of 2) ± 1) se puede hacer en un ciclo
- Ejemplo: r0 = r1 * 5
  = r1 + (r1 * 4)

  ï ADD r0, r1, r1, LSL #2
- Ejemplo: r2 = r3 * 105
  = r3 * 15 * 7
  = r3 * (16 - 1) * (8 - 1)

  ï RSB r2, r3, r3, LSL #4 ; r2 = r3 * 15
  ï RSB r2, r2, r2, LSL #3 ; r2 = r2 * 7

Segundo Operando: Valor Inmediato(1)

- No existe una instrucción que cargue una constant inmediata de 32 bit en un registro sin realizar una carga de dato desde memoria.
  - Todas las instrucciones ARM son de 32 bits
  - Las instrucciones ARM no usan el stream de la instrucción como dato.
- El formato de la instrucción de procesamiento de dato tiene 12 bits disponible para el operand2
  - Si se usa directamente nos da un rango de 4096.
- Sin embargo, se usa para almacenar constants de 8 bits, lo que da un rango de 0 - 255.
- Estos 8 bits se pueden rotar a la derecha a través de un número par de posiciones (ie RORs por 0, 2, 4,..30).
  - Esto da un rango mucho mayor de constants que se pueden cargar directamente, aunque algunas constants siempre tendrán que cargarse desde memoria.

Segundo Operando: Valor Inmediato(2)

- Esto nos da:
  - 0 - 255 [0 - 0xff]
  - 256,260,264,..,1020 [0x100-0x3fc, step 4, 0x40-0xff ror 30]
  - 1024,1040,1056,..,4080 [0x400-0xff0, step 16, 0x40-0xff ror 28]
  - 4096,4160, 4224,..,16320 [0x1000-0x3fc0, step 64, 0x40-0xff ror 26]
- Estos se pueden cargar usando, por ejemplo:
  - MOV r0, #0x40, 26 ; => MOV r0, #0x1000 (ie 4096)
- Para hacerlo más fácil, el assembler lo convertirá a esta forma para nosotros si simplemente se entrega la constant deseada:
  - MOV r0, #4096 ; => MOV r0, #0x1000 (ie 0x40 ror 26)
- El complement bi-a-bit se puede usar con MVN:
  - MOV r0, #0xFFFFFFFF ; assembles to MVN r0, #0
- Si la constant deseada no se puede generar, se reporta un error

Cargando Constantes de Completos 32 Bits

- Aunque el mecanismo de MOV/MVN carga un rango grande de constants en un registro, algunas veces este mecanismo no puede generar la constant deseada.
- Por lo tanto, el assembler proporciona un método para cargar *ANY* constante de 32 bits:
  - `LDR rd,=numeric constant`
- Si la constante se puede construir usando un MOV o un MVN entonces ésta será la instrucción realmente generada.
- De otra manera, el assembler producirá una instrucción LDR con una dirección de PC-relativo para leer la constante desde un pool.
  - `LDR r0,=0x42 ; generates MOV r0,#0x42`
  - `LDR r0,=0x55555555 ; generate LDR r0,[pc, offset to lit pool]`
- Como este mecanismo siempre generará la mejor instrucción para un caso dado, es la forma recomendada para cargar constantes.

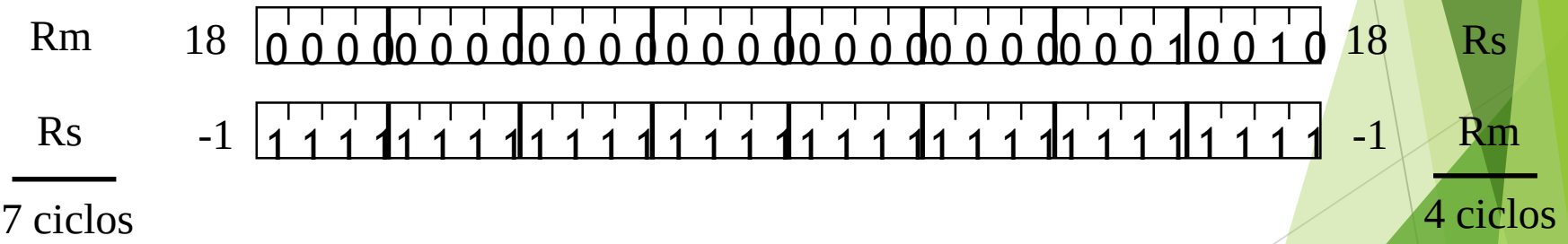
Instrucciones de Multiplicación

- El ARM básico proporciona instrucciones de multiplicación.
- Multiplicación
  - MUL{<cond>}{S} Rd, Rm, Rs ; Rd = Rm * Rs
- Multiplicación acumulada - entrega la suma sin costo de tiempo
  - MLA{<cond>}{S} Rd, Rm, Rs,Rn ; Rd = (Rm * Rs) + Rn
- Restriciones de uso:
  - Rd and Rm no pueden ser el mismo registro
    - Se puede evitar intercambiando Rm y Rs. Esto funciona porque la multiplicación es conmutativa.
  - No se puede usar el PC.

  Esto será denunciado por el ensamblador si no se toma en cuenta.
- Los operandos se pueden considerer con Signo o sin Signo
  - Es el usuario quién debe interpretarlo correctamente.

Implementación de la Multiplicación

- ARM hace uso del algoritmo de Booth's para realizar la multiplicación con enteros
- En ARMs no-M, esto opera con 2 bits de Rs a la vez.
  - Para cada par de bits esto toma 1 ciclo (más un ciclo para comenzar).
  - Sin embargo, cuando no quedan más 1's en Rs, la multiplicación termina tempranamente.
- Ejemplo: Multiplique 18 y -1 : Rd = Rm * Rs



- Nota: El compilador no usa el criterio de terminación temprana para decidir en que orden pone los operandos.

Instrucciones de Multiplicación Extendidas

- M variantes de núcleos ARM contienen hardware para multiplicación extendida. Esto proporciona tres mejoras:
  - Se usa un algoritmo de ***Booth's de 8 bits***
    - La multiplicación se realiza más rápido(el máximo para instrucciones estándar es 5 ciclos)
  - ***El método de terminación temprana*** complete la multiplicación cuando todos los bits (set) remanentes tienen
    - Todos ceros (tal como con no-M ARMs), o
    - Todos uno.

    Así el ejemplo anterior termina tempranamente en 2 ciclos en ambos casos
  - ***Se pueden producir resultados 64 bit*** desde dos operandos de 32 bits
    - Alta exactitud.
    - Un par de registros se usa para almacenar el resultado.

Multiplicación-Long y Multiplicación-Acumulada Long

- Las instrucciones son
  - MULL la cual da RdHi,RdLo:=Rm*Rs
  - MLAL la cual da RdHi,RdLo:=(Rm*Rs)+RdHi,RdLo
- Sin embargo, el resultado de 64 bits ahora importa(instrucciones de multiplicación de baja precision simplemente descartan los 32 bits de arriba(top))
  - Se necesita especificar si los operandos son con signo o sin signo
- Por lo tanto, la sintaxis de las nuevas instrucciones es:
  - UMULL{<cond>}{S} RdLo,RdHi,Rm,Rs
  - UMLAL{<cond>}{S} RdLo,RdHi,Rm,Rs
  - SMULL{<cond>}{S} RdLo, RdHi, Rm, Rs
  - SMLAL{<cond>}{S} RdLo, RdHi, Rm, Rs
- No generado por el compilador.

Warning : No predecible en no-M ARMs.

Instrucciones Load / Store

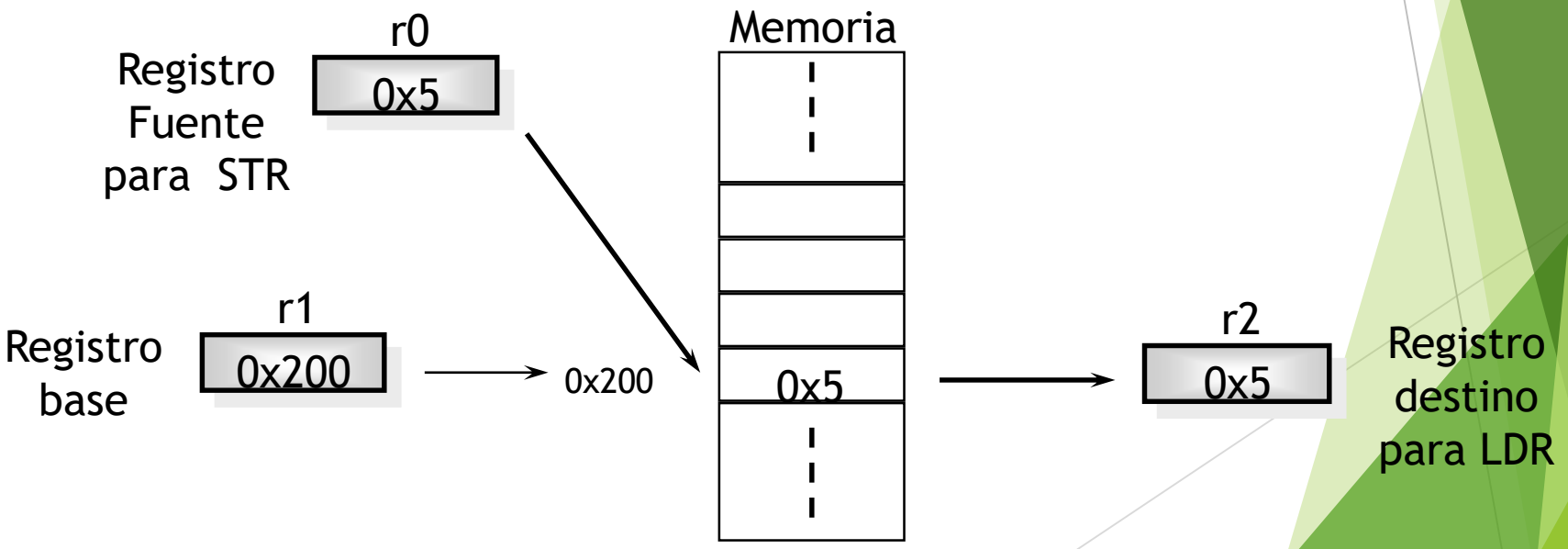
- ARM es una arquitectura Load / Store :
  - No soporta operaciones memoria a memoria.
  - Debe mover los valores a registros antes de usarlos.
- Esto puede sonar ineficiente, pero en la práctica no lo es:
  - Load carga valores desde memoria a registros.
  - Procesa los datos en registros usando varias instrucciones de procesamiento de datos, las que no son retrasadas por el acceso a memoria.
  - Store pasa resultados de registros a memoria.
- ARM tiene tres conjuntos de instrucciones las que interactúan con la memoria
  - Transferencia de datos a un registro (LDR / STR).
  - Transferencia de un bloque de datos (LDM/STM).
  - Intercambio de un dato (SWP).

Transferencia de un Registro

- Las instrucciones básicas load y store son:
  - Load y Store Word o Byte
    - LDR / STR / LDRB / STRB
- La arquitectura ARM en Version 4 también posee soprte para halfwords y datos con signo.
  - Halfword Load y Store
    - LDRH / STRH
  - Load Signed Byte o Halfword – valor load y signo extendido a 32 bits.
    - LDRSB / LDRSH
- Todas estas instrucciones se pueden ejecutar condicionalmente insertando el bit de condición en STR / LDR.
  - e.g. LDREQB
- Sintaxis:
  - <LDR|STR>{<cond>}{<size>} Rd, <address>

Load y Store, Word o Byte: Registro Base

- La ubicación de memoria a ser accesada se mantiene en un registro base
  - STR r0, [r1] ; Almacena contenido de r0 a una ubicación apuntada
    ; por el contenido de r1.
  - LDR r2, [r1] ; Carga r2 con el contenido de una ubicación de memoria
    ; apuntado por el contenido de r1.

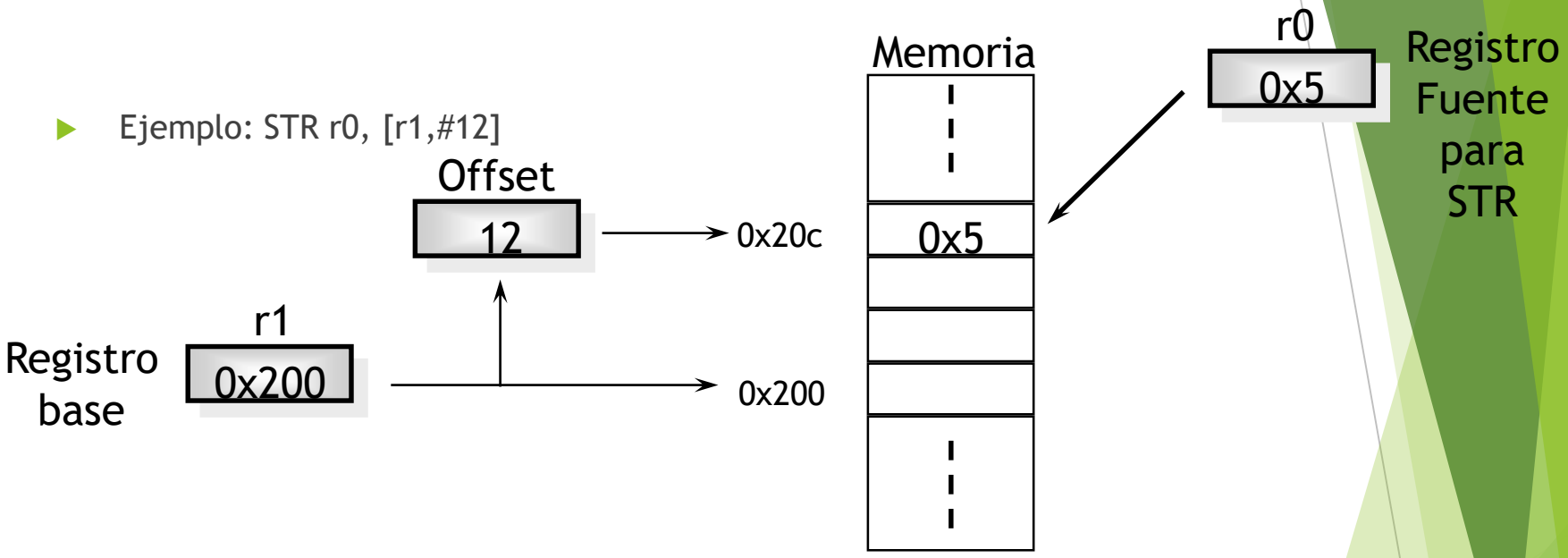


Load y Store, Word o Byte: Offsets desde el Registro Base

- Además de accesar la ubicación actual contenida en el registro base, estas instrucciones pueden accesar una ubicación a una distancia(offset) desde el puntero del registro base.
- Este offset puede ser
  - Un valor inmediato sin signo de 12 bits(ie 0 - 4095 bytes).
  - Un registro, opcionalmente desplazado por un valor inmediato
- Esto puede ser o sumado o restado desde el registro base:
  - Prefijar el valor del offset o registro con '+' (default) or '-'.
- Este offset se puede aplicar:
  - Antes de hacer la transferencia: direccionamiento ***Pre-indexed***
    - opcionalmente *auto-incrementing* el registro base, mediante postfixing la instrucción con un'!'.
  - Después de hacer la transferencia: direccionamiento ***Post-indexed***
    - Causando que el registro base sea *auto-incremented*.

Load y Store, Word o Byte: Direccionamiento Pre-indexado

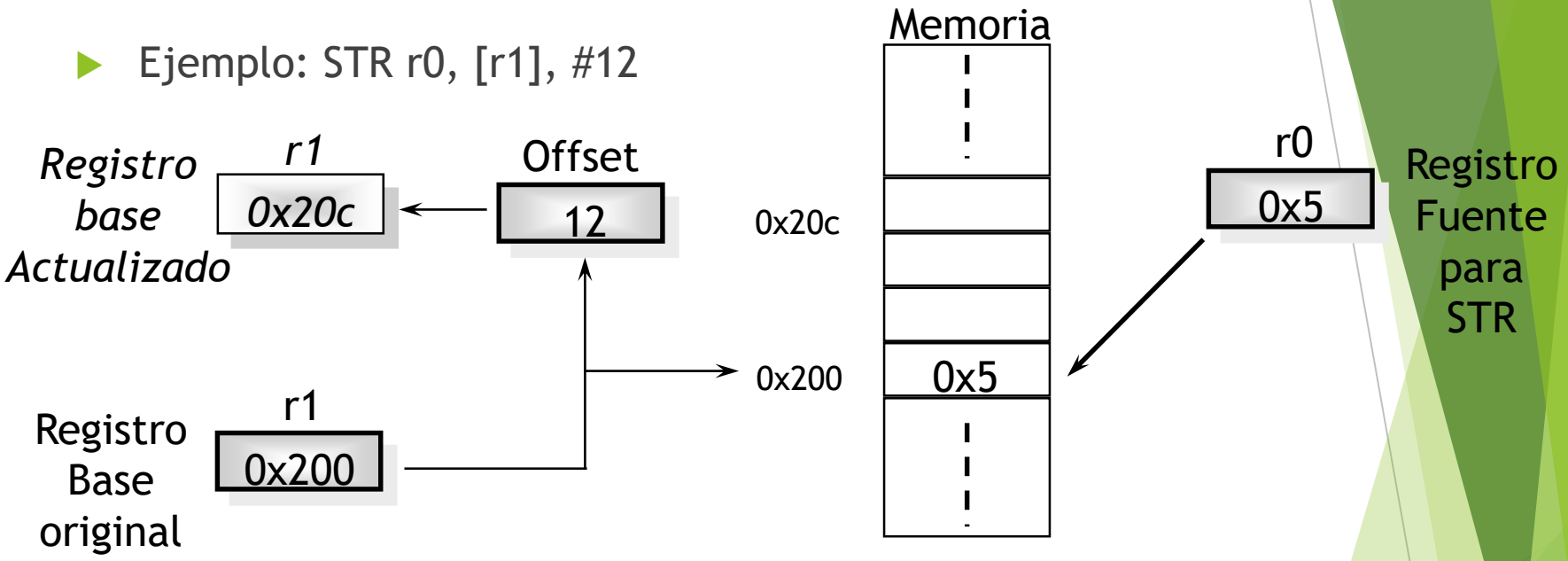
- Ejemplo: STR r0, [r1,#12]



- Para almacenar en la ubicación 0x1f4 use mejor: STR r0, [r1,#-12]
- Para auto-incrementar el puntero base a 0x20c use: STR r0, [r1, #12]!
- Si r2 contiene 3, accese 0x20c multiplicando esto por 4:
  - STR r0, [r1, r2, LSL #2]

Load y Store, Word o Byte: Direccionamiento Post-indexado

- Ejemplo: STR r0, [r1], #12



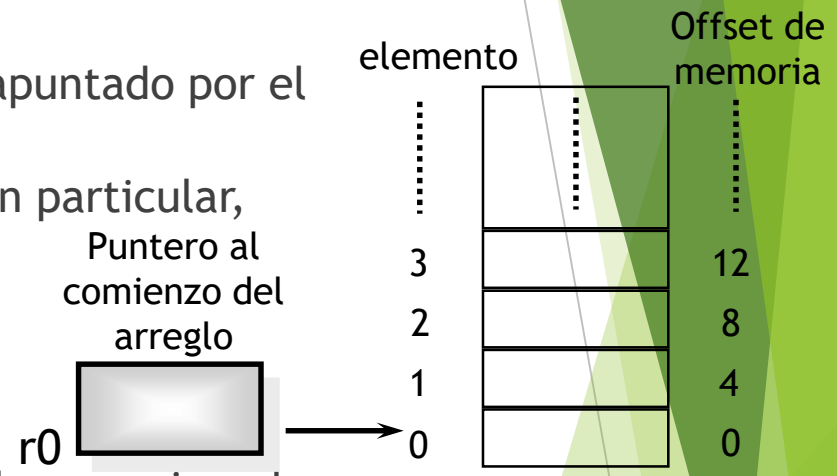
- Para auto-incrementar el registro base a la ubicación 0x1f4 use mejor:
  - STR r0, [r1], #-12
- Si r2 contiene 3, auto-incremente el registro base a 0x20c multiplicando esto por 4:
  - STR r0, [r1], r2, LSL #2

Load y Stores con Modo Usuario Priviligeado

- Cuando se usa direccionamiento post-indexado, existe otra forma de Load/Store Word/Byte:
  - <LDR|STR>{<cond>}{B}**T** Rd, <post_indexed_address>
- Cuando se usa en modo priviligeado, esto hace el load/store con modo priviligeado.
  - Normalmente usado por un manejador de excepciones que emula una instrucción de acceso a memoria que podría ejecutarse en modo normal.

Ejemplo: Uso de Modos de Direccionamiento

- Imagine un arreglo, el primer element es apuntado por el contenido del registro r0.
- Si queremos tener acceso a un elemento en particular, usamos direccionamiento pre-indexado:
  - r1 es el elemento que queremos.
  - LDR r2, [r0, r1, LSL #2]



- Si queremos ir por cada element del arreglo, por ejemplo para sumar todos los elementos en el arreglo, entonces usamos direccionamiento pos-indexado en un loop:
  - r1 es la dirección del element actual(inicialmente igual a r0).
  - LDR r2, [r1], #4

Use otro registro para almacenar la dirección del elemento final,
para que el loop se termine correctamente.

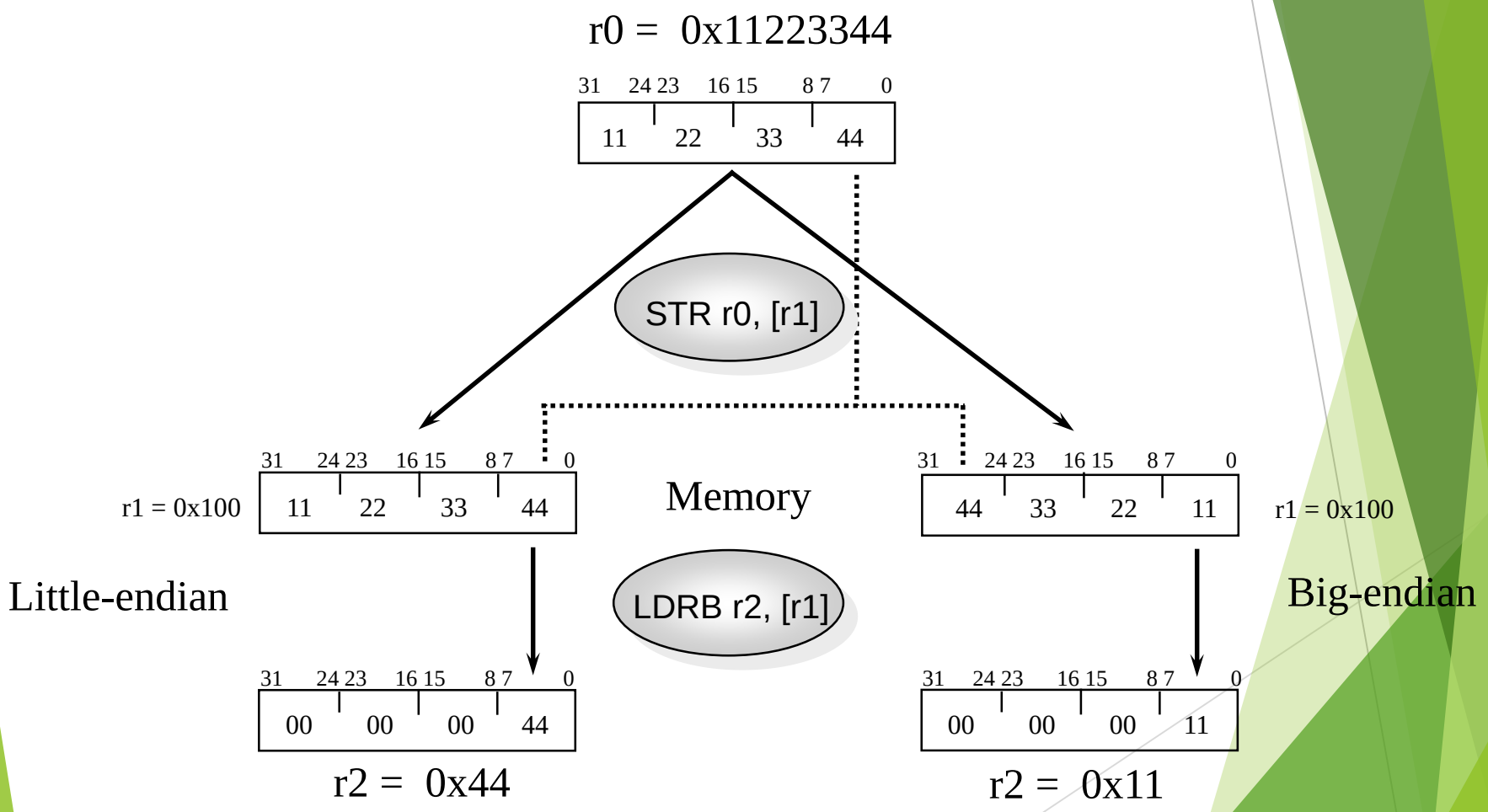
Offsets para Halfword y Halfword con signo / y acceso de Byte

- La halfword Load y Store y Load Signed Byte o instrucciones Halfword pueden hacer uso de direccionamiento pre- y post-indexados en prácticamente la misma forma que las instrucciones básicas load y store.
- Sin embargo, los formatos de offsets actuales son más restringidos:
  - El valor inmediato está limitado a 8 bits(y no 12 bits) lo que da un offset de 0-255 bytes.
  - A la forma registro no se le puede aplicar un shift.

El Efecto de endianess

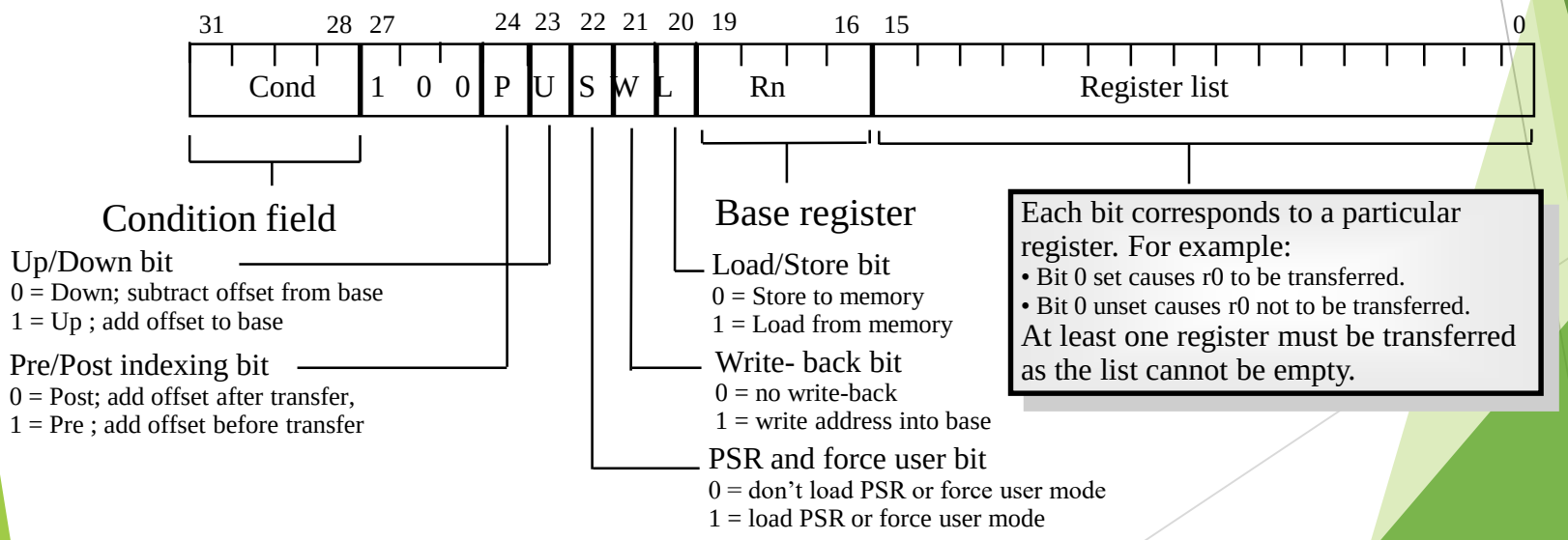
- ARM sepuede fijar para accesar los datos en formato little o big endian.
- Little endian:
  - Byte menos significativo de es almacenado en ***bits 0-7*** de una dirección de palabra.
- Big endian:
  - Byte menos significativo es almacenado en ***bits 24-31*** de una dirección de palabra.
- Esto no tiene relevancia a menos que los datos sean almacenados como palabras y después accesados en pequeñas cantidades (halfwords o bytes).
  - Cual byte / halfword sea accesado dependerá del endianess del Sistema involucrado

Ejemplo de Endianess



Transferencia de Datos en Bloques(1)

- Las múltiples instrucciones Load and Store (LDM / STM) permiten entre 1 y 16 registros ser transferidos a o desde la memoria.
- Los registros transferidos pueden ser:
  - Cualquier subconjunto del banco de registros (default).
  - Cualquier subconjunto del banco de registros en modo usuario cuando se está en modo priviligeado (postfix la instrucción con un '^').

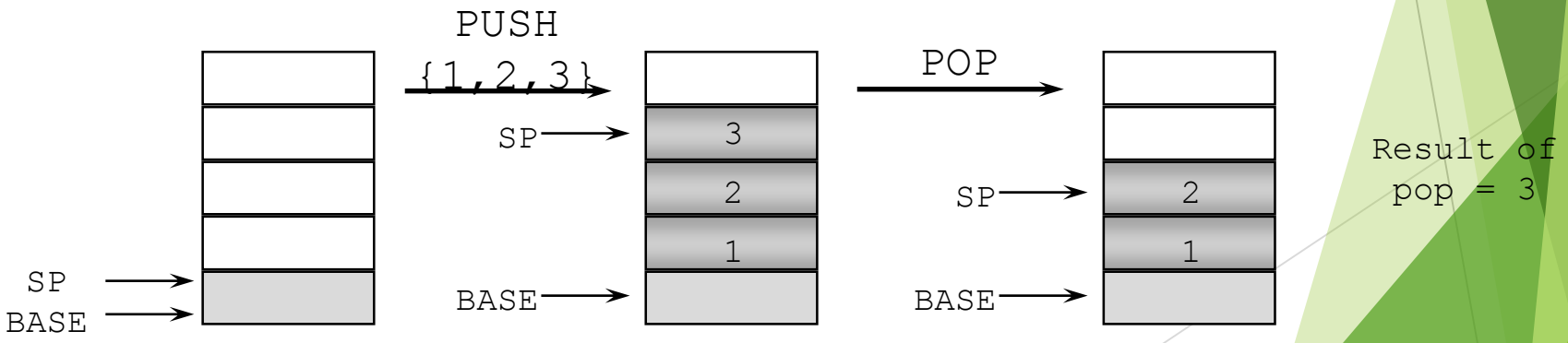


Transferencia de Bloques de Datos(2)

- El registro base se usa para determiner dónde debe ocurrir el acceso a memoria
  - 4 modos diferentes de direccionamiento permiten incremento o decremento inclusivo o exclusivo de la ubicación del registro base.
  - El registro base se puede, opcionalmente, actualizar después de la transferencia (agregándole un '**!**').
  - El número de registro más bajo siempre se transfiere a/desde la ubicación menor de memoria.
- Estas instrucciones son muy eficientes para
  - Salvar y restaurar contexto
    - Útil para mirar la memoria como Stack.
  - Mover grandes bloques de datos en la memoria
    - Útil para representar la funcinalidad de las instrucciones.

Stacks

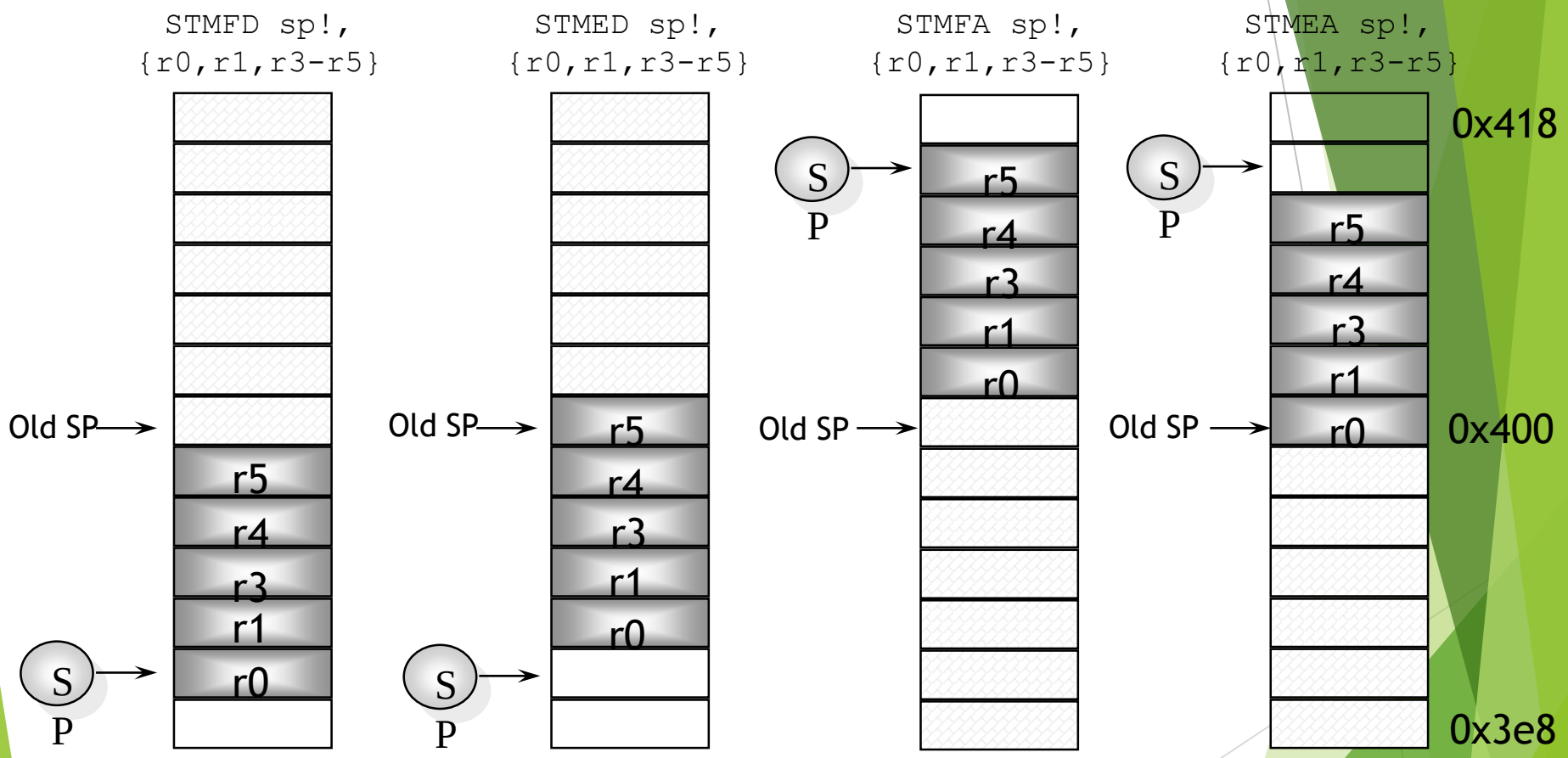
- Un stack es un área de la memoria la cual crece cuando nuevos datos son “pushed”(empujados) en el “top”(tope) de ella, y disminuye cuando los datos son “popped”(extraídos) fuera del tope.
- Dos punteros definen los límites Corrientes del Stack.
  - Un punter base
    - Usado para apuntar al fondo del stack (la primera ubicación).
  - Un puntero de stack
    - Usado para apuntar al tope actual del stack.



Operación del Stack

- Tradicionalmente, un stack crece hacia abajo en la memoria, con el último valor empujado a la dirección más baja. ARM también soporta stacks ascendentes , donde la estructura del stack crece hacia arriba en la memoria
- El valor del punter del stack puede ser:
  - Apunta a la última dirección ocupada(stack completo)
    - Y por lo tanto necesita pre-decrementar(ie. antes de empujar “push”)
  - Apunta a la próxima dirección ocupada(stack vacío)
    - Y por lo tanto necesita post-decrementar(ie. después de empujar)
- El tipo de stack a ser usado está dado por el postfix de la instrucción:
  - STMFD / LDMFD : Full Descending stack
  - STMFA / LDMFA : Full Ascending stack.
  - STMED / LDMED : Empty Descending stack
  - STMEA / LDMEA : Empty Ascending stack
- Nota: El compilador ARM siempre usará un Full descending stack.

Ejemplos de Stack



Stacks y Subrutinas

- Un uso de los stack es crear espacio de trabajo temporal para los registros para subrutinas. Cualquier registro que se necesite empujar al stack al comienzo de la subrutina y extraer de nuevo al final para restaturar antes del retorno del llamado:

```
STMFD sp!,{r0-r12, lr}     ; stack all registers
........            ; and the return address
........
LDMFD sp!,{r0-r12, pc}     ; load all the registers
                ; and return automatically
```

- Si la instrucción pop tien el bit ‘S’ bit set (usando ‘^’) entonces la transferencia del PC en modo priviligeado ,causará que el SPSR sea copiado al CPSR.

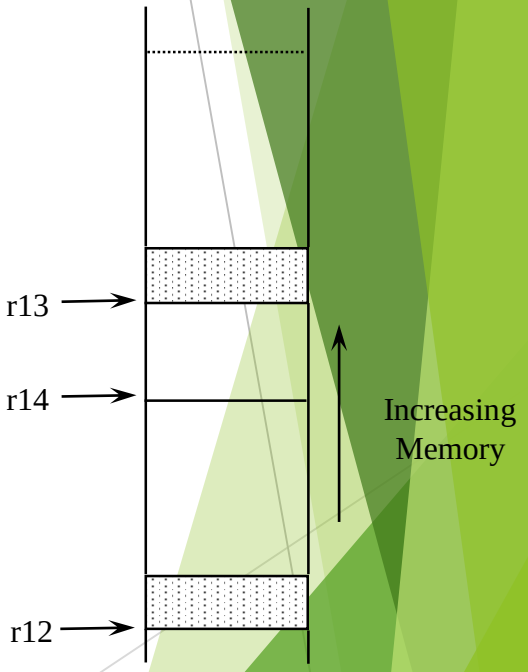
Funcionalidad Directa de la Transferencia de Bloques de Datos

- Cuando LDM / STM no se están usando para implementar stacks, hay que especificar exactamente la funcionalidad de la instrucción:
  - i.e. especificar si incrementar / decrementar el punter base, antes o después del acceso a memoria
- Con el fin de hacer esto, LDM / STM posee una sintaxis adicional a la del stack:
  - STMIA / LDMIA : Incrementar After
  - STMIB / LDMIB : Incrementar Before
  - STMDA / LDMDA : Decrementar After
  - STMDB / LDMDB : Decrementar Before

Ejemplo: Copia de un Block

- Copiar un bloque de memoria, el cual es un múltiplo exacto de 12 palabras desde la ubicación apuntada por r12 a la ubicación apuntada por r13. r14 apunta al final del bloque copiado.

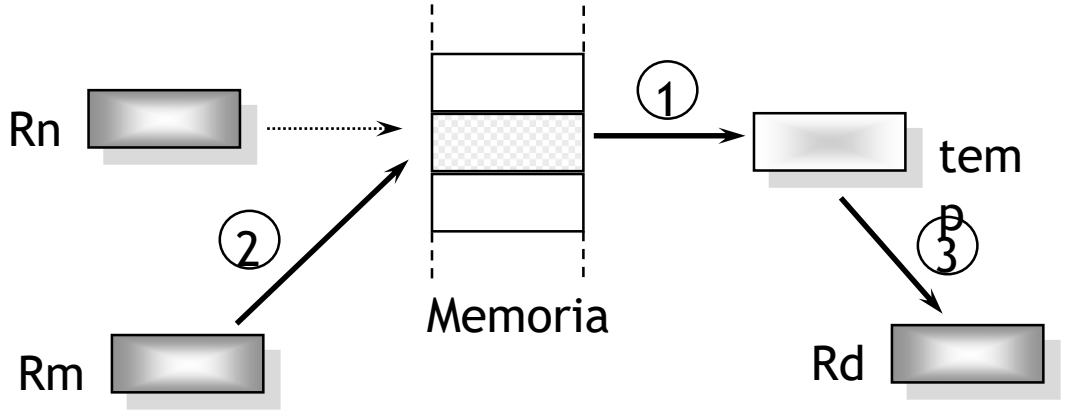
```
; r12 points to the start of the source data
; r14 points to the end of the source data
; r13 points to the start of the destination data
loop    LDMIA   r12!, {r0-r11} ; load 48 bytes
    STMIA   r13!, {r0-r11} ; and store them
    CMP r12, r14    ; check for the end
    BNE loop        ; and loop until done
```



- Este loop transfiere 48 bytes en 31 ciclos
- Sobre 50 Mbytes/sec a 33 MHz

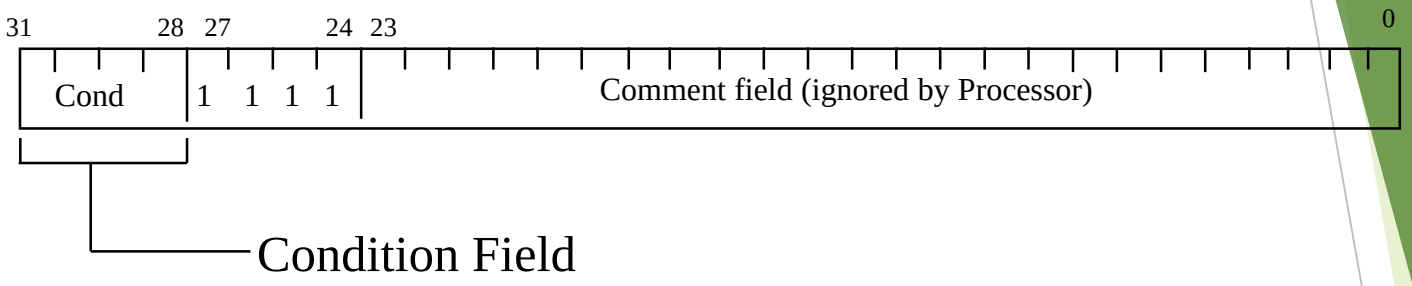
Instrucciones Swap y Swap Byte

- Operaciones atómicas de una lectura de memoria seguida por una escritura a memoria la cual mueve byte o word entre registros y memoria
- Sintaxis:
  - SWP{<cond>}{B} Rd, Rm, [Rn]



- Así, para implementar un swap real de contenidos haga Rd = Rm.
- El compilador no puede generar esta instrucción.

Interrupción por Software (SWI)



- En efecto, SWI es una instrucción definida por el usuario.
- Causa una excepción al vector de hardware de SWI(obliga el cambio a modo supervisor, con el correspodiente estado salvado), para luego llamar al manejador del SWI.
- El manejador puede examinar el campo de comentario de la instrucción para ver que operación fue solicitada
- Usando el mecanismo SWI, un SO puede implementar un conjunto de operaciones que pueden ser llamadas desde modo usuario.