ELPL Embedded Low-Power Laboratory

Embedded System Application

4190.303C

2010 Spring Semester

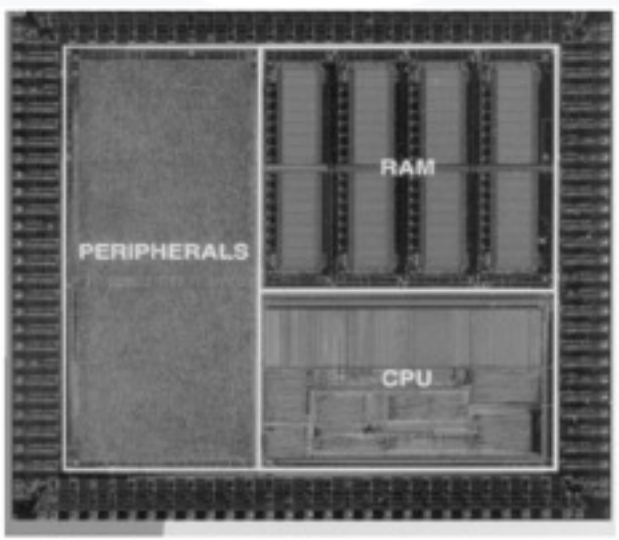
ARM Processor Core

Naehyuck Chang
Dept. of EECS/CSE
Seoul National University
naehyuck@snu.ac.kr

2010년 3월 8일 월요일

ARM Processor Core - Introduction

- Architecture
  - Versions 1 and 2 – Acorn RISC, 26-bit address
  - Version 3 – 32-bit address, CPSR, and SPSR
  - Version 4 – half-word, Thumb (compressed instruction)
  - Version 5 – Digital Signal Processing, Java byte code Extensions
  - Version 6 – SIMD, Thumb-2, Multiprocessing
  - Version 7 – extended SIMD, improved floating point support
- Processor cores
  - ARM7TDMI (Thumb, debug, multiplier, ICE) – version 4T, low-end ARM core, 3-stage pipeline
  - ARM9TDMI – 5-stage pipeline
  - ARM10TDMI – version 5
  - ARM11 - 8-stage pipeline, version 6
  - Cortex - version 7
- CPU Core: co-processor, MMU, AMBA
  - ARM 710, 720, 740
  - ARM 920, 940
  - ARM11 MPCore, Cortex -A,R,M Series



ELPL Embedded Low-Power Laboratory

2010년 3월 8일 월요일

RISC and CISC microprocessors

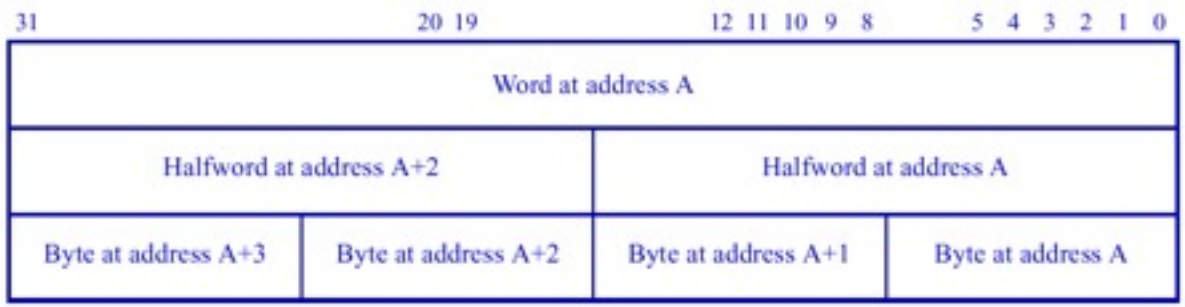
- RISC: Reduced Instruction Set Computer
- CISC : Complex Instruction Set Computer
  - 80% programs use only 20% of instructions
  - Reduce instructions and emulate infrequently used instructions with multiple instructions
- Reduced instructions → simple hardware → fast operation
  - Mostly 1 clock per instruction
    - Equal instruction length
    - External bus width and internal bus width must be the same
    - Memory access is allowed only for load/store instructions
    - Arithmetic and logical operations are done by registers
    - No microprograms for fast operation
    - Use of registers is a primary concern of performance: compiler technology

32 bit RISC's Data Types

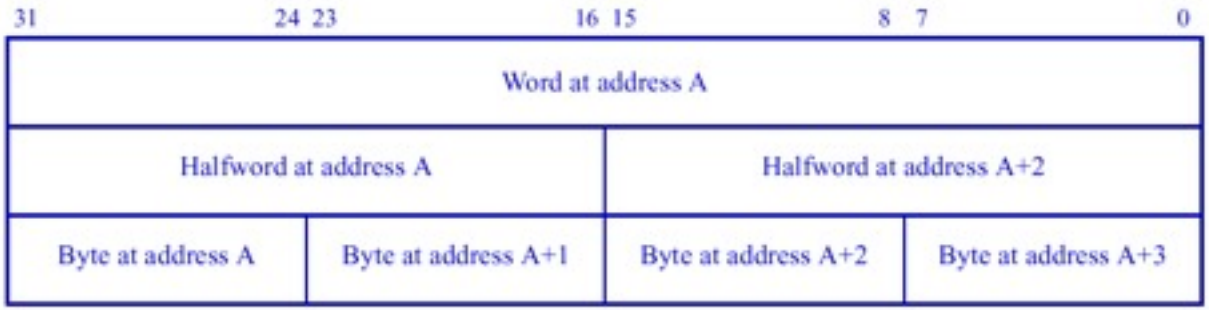
- Byte, halfword, and word (aligned)
  - Signed and unsigned integers
  - All data operations are performed on word quantities
  - Load and store operations transfer bytes, halfwords, and words to and from memory (zero- or sign-extending)
  - ARM instructions are exact one word and aligned on a 4-byte boundary (Thumb instructions are exact one halfword)
- Memory and address
  - A flat space of 2^32 bytes
  - Little-endian (1st byte is the least significant byte) or big-endian (1st byte is the most significant byte)
  - ARM can support either one or both (need a hardware input to configure the endianness)

Effect of Endianess

- Little endian: Least significant byte of a word is stored in bits 0-7 of an addressed word.

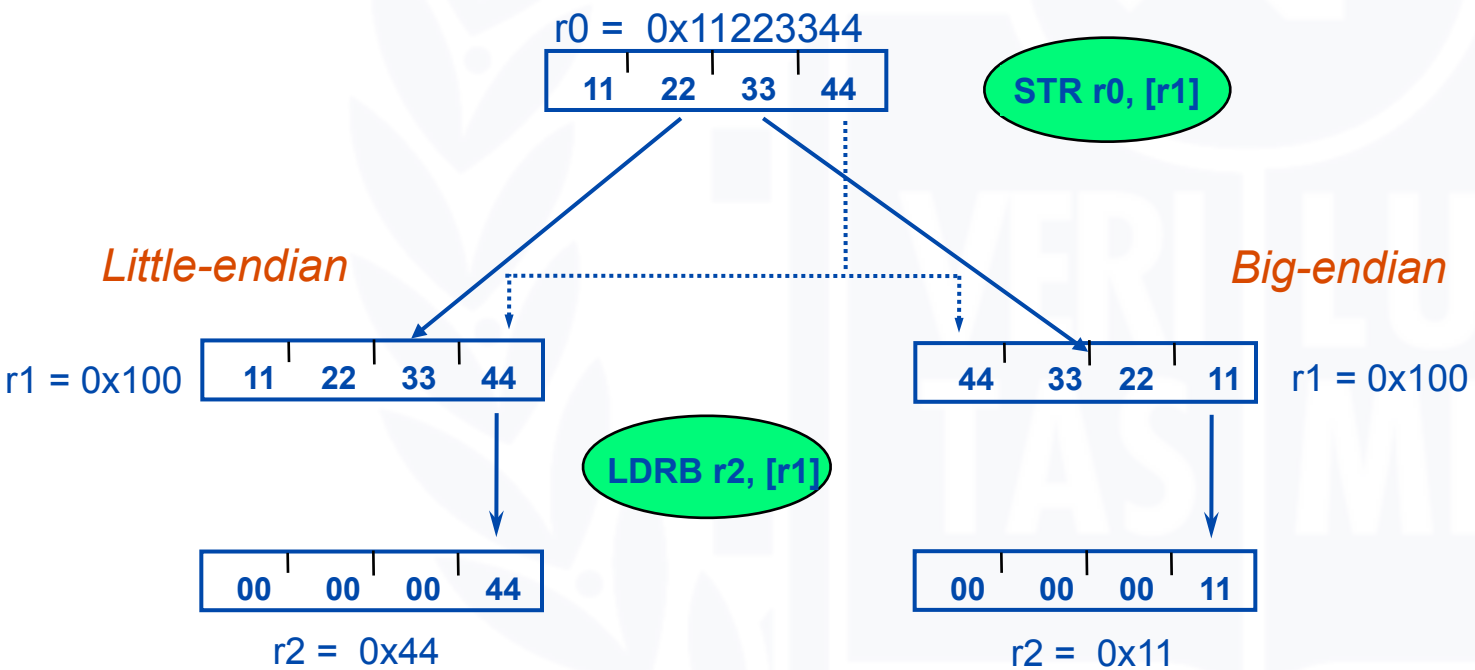


- Big endian: Least significant byte of a word is stored in bits 24-31.



Endianess Example

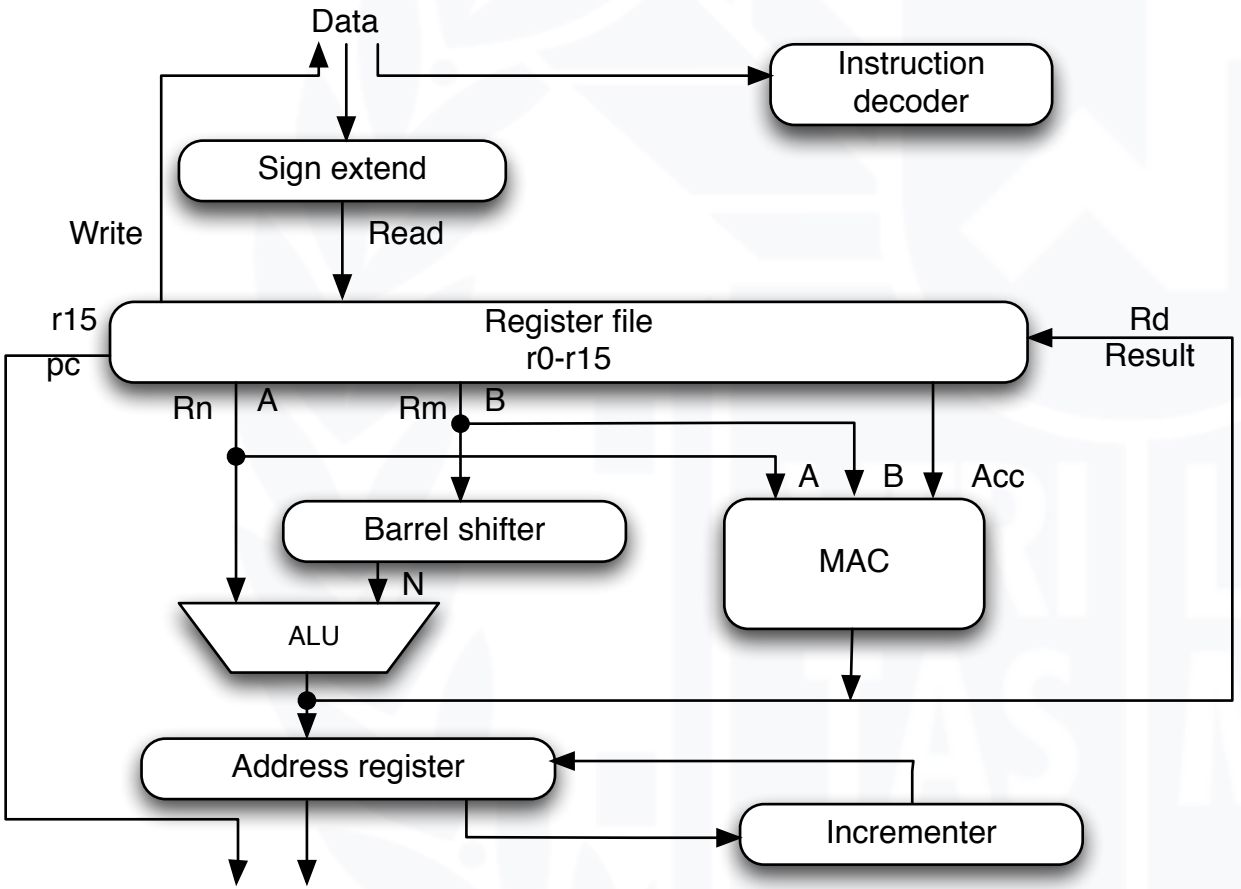
- This has no real relevance unless data is stored as words and then accessed in smaller sized quantities (halfwords or bytes).
  - Which byte / halfword is accessed will depend on the endianess of the system involved.



ELPL Embedded Low-Power Laboratory

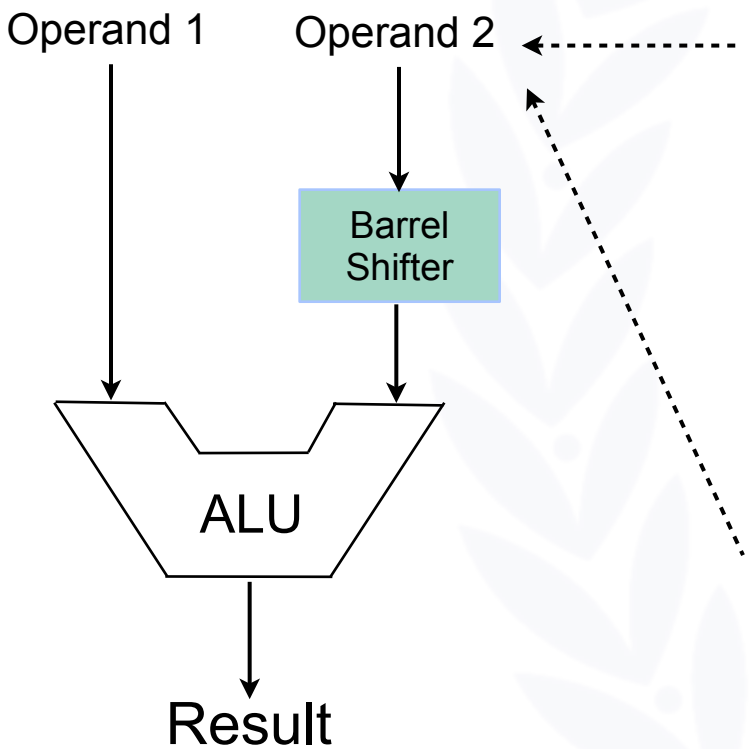
ARM core

- ARM core dataflow model



The Barrel Shifter

- ARM has a barrel shifter which provides a mechanism to carry out shifts as part of other instructions.

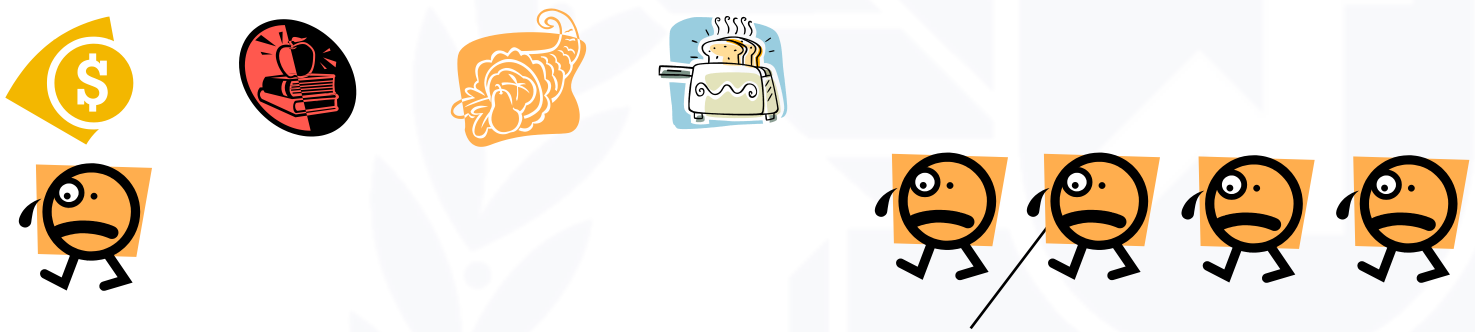


- Register, optionally with shift operation applied.
- Shift value can be either be:
  - 5 bit unsigned integer
  - specified in bottom byte of another register.

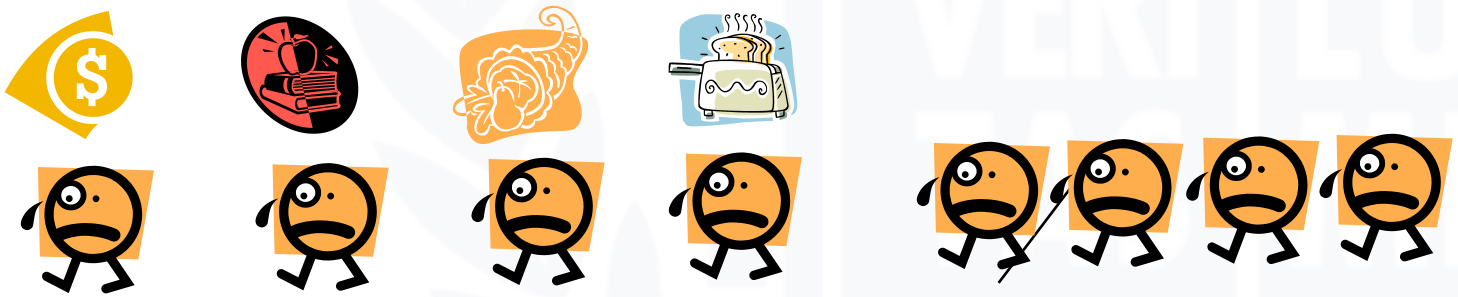
- Immediate value
  - can be rotated right through an even number of positions.
  - assembler will calculate rotate for you from constant.

Pipeline operation

- Non-pipelined operation

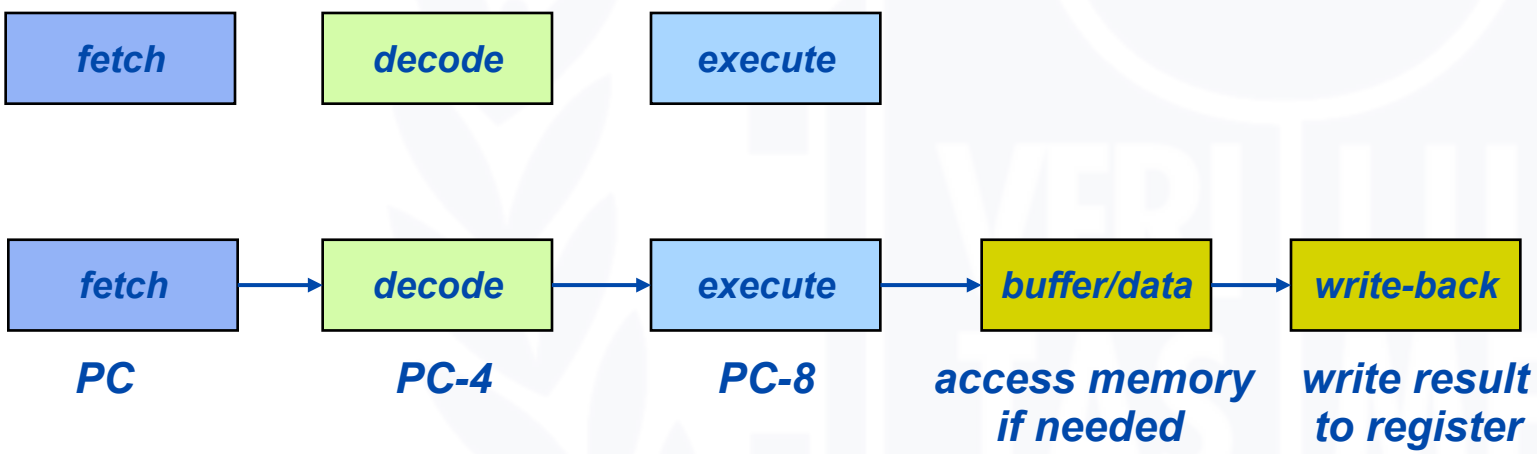


- Pipelined operation



The Instruction Pipeline

- The ARM uses a pipeline in order to increase throughput (the speed of the flow of instructions to the processor)
  - PC points to the instruction being fetched
- 3 stages (ARM7) and 5 stages (ARM9TDMI)



Processor Modes

- True multi-user systems
- Mode changes may be made under software control or may be caused by external interrupts or exception processing.
- Most application programs will execute in user mode.
- Other privileged modes will be entered to service interrupts or exceptions or to access protected resources:

Processor mode			Description
1	User	(usr)	the normal program execution mode
2	FIQ	(fiq)	designed to support a high-speed data transfer or channel process
3	IRQ	(irq)	used for general-purpose interrupt handling
4	Supervisor	(svc)	a protected mode for the operating system
5	Abort	(abt)	used to implement virtual memory and/or memory protection
6	Undefined	(und)	used to support software emulation of hardware coprocessors
7	System	(sys)	used to run privileged operating system tasks (Architecture Version 4 only)

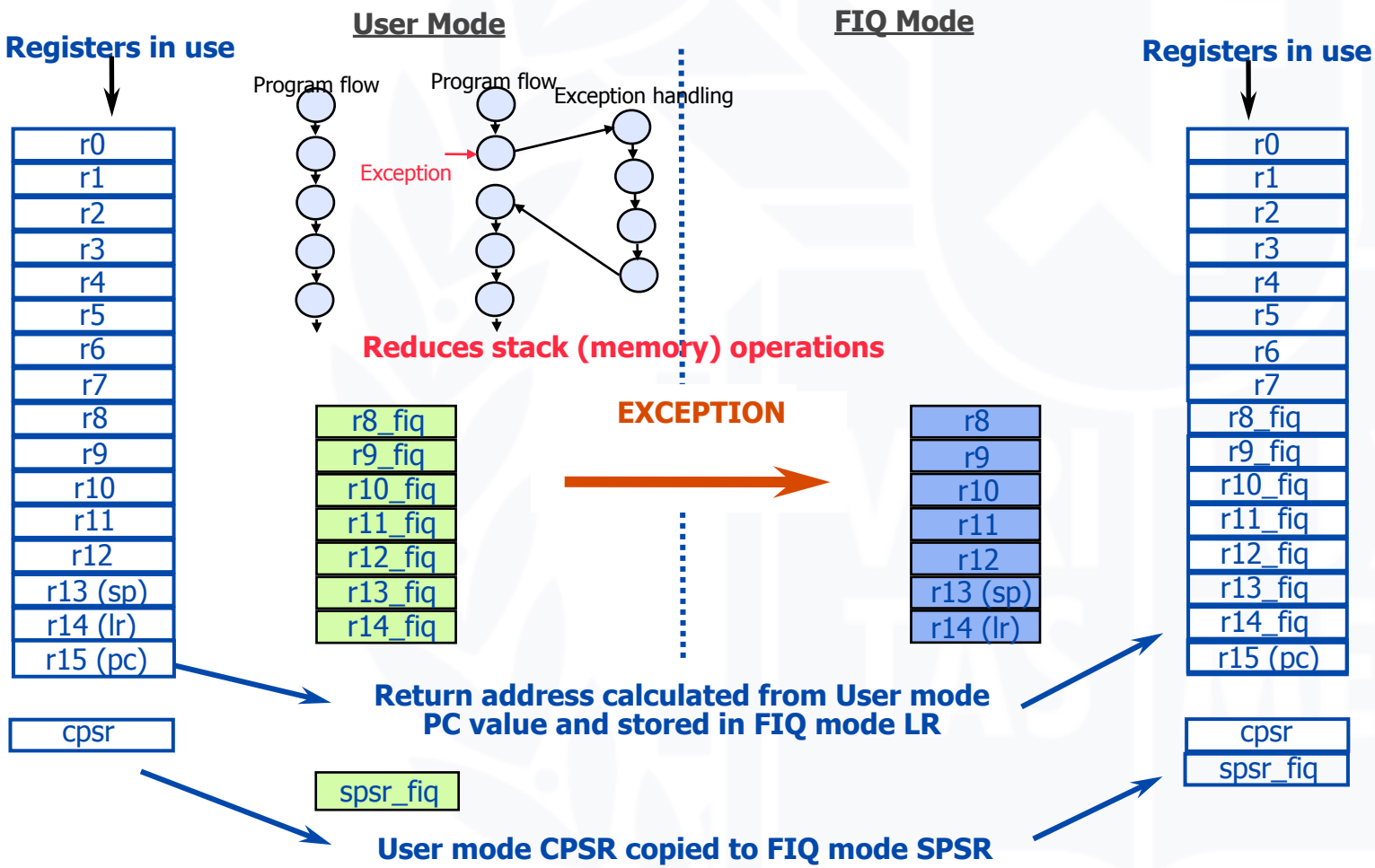
Register Organization

- Many registers
  - Reduce memory access
  - Difficult to fully utilize
- Register banks
  - Registers are arranged into several banks, being governed by the processor mode
  - Reduce mode switching overhead
- Each mode can access
  - A particular set of R0-R12 registers
  - A particular R13 (SP: stack pointer) and R14 (LR: link register)
  - R15 (the program counter)
  - CPSR (the current program status register)
  - Privileged modes can also access a particular SPSR (saved program status register)

User/ System	Supervi-sor	Abort	Undefined	Interrupt	Fast interrupt
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	R8	R8	R8	R8_FIQ
R9	R9	R9	R9	R9	R9_FIQ
R10	R10	R10	R10	R10	R10_FIQ
R11	R11	R11	R11	R11	R11_FIQ
R12	R12	R12	R12	R12	R12_FIQ
R13	R13_SVC	R13_ABORT	R13_UNDEF	R13_IRQ	R13_FIQ
R14	R14_SVC	R14_ABORT	R14_UNDEF	R14_IRQ	R14_FIQ
PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_SVC	SPSR_ABORT	SPSR_UNDEF	SPSR_IRQ	SPSR_FIQ

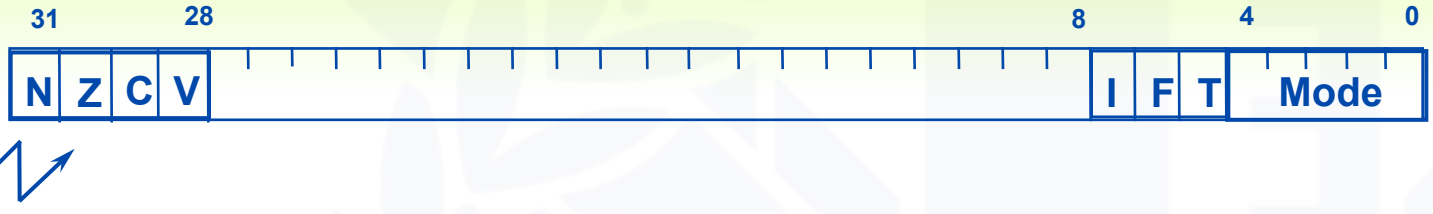
ELPL Embedded Low-Power Laboratory

Register Example: User to FIQ Mode



ELPL Embedded Low-Power Laboratory

Program Status Registers (CPSR & SPSRs)



Copies of the ALU status flags (latched if the instruction has the "S" bit set).

- Condition Code Flags
  - N = Negative result from ALU flag.
  - Z = Zero result from ALU flag.
  - C = ALU operation Carried out
  - V = ALU operation oVerflowed
- Interrupt Disable bits.
  - I = 1, disables the IRQ.
  - F = 1, disables the FIQ.
  - T Bit: Processor in ARM (0) or Thumb (1)
- Mode Bits: processor mode

M[4:0]	Mode
10000	User
10001	FIQ
10010	IRQ
10011	SVC
10111	Abort
11011	Undef
11111	System

ELPL Embedded Low-Power Laboratory

The Program Counter (R15)

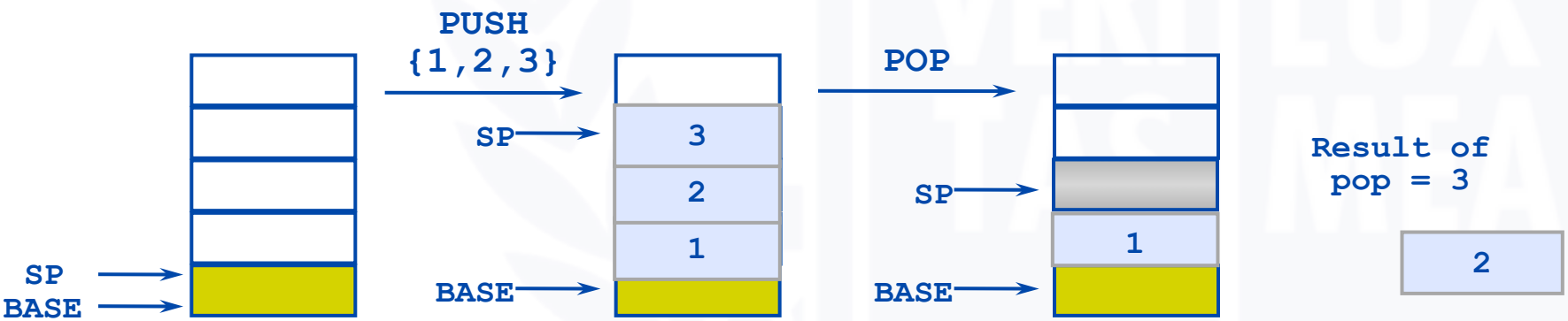
- When the processor is executing in ARM state:
  - All instructions are 32 bits in length
  - All instructions must be word aligned
  - Therefore the PC value is stored in bits [31:2] with bits [1:0] equal to zero (as instruction cannot be halfword or byte aligned).
- R14 is used as the subroutine link register (LR) and stores the return address when Branch with Link (BL) operations are performed, calculated from the PC.
- Thus to return from a linked branch

```
MOV r15, r14
MOV pc,lr
```

Stacks

- A stack is an area of memory which grows as new data is "pushed" onto the "top" of it, and shrinks as data is "popped" off the top.

- Two pointers define the current limits of the stack.
  - A base pointer (frame pointer): used to point to the "bottom" of the stack (the first location).
  - A stack pointer: used to point the current "top" of the stack.

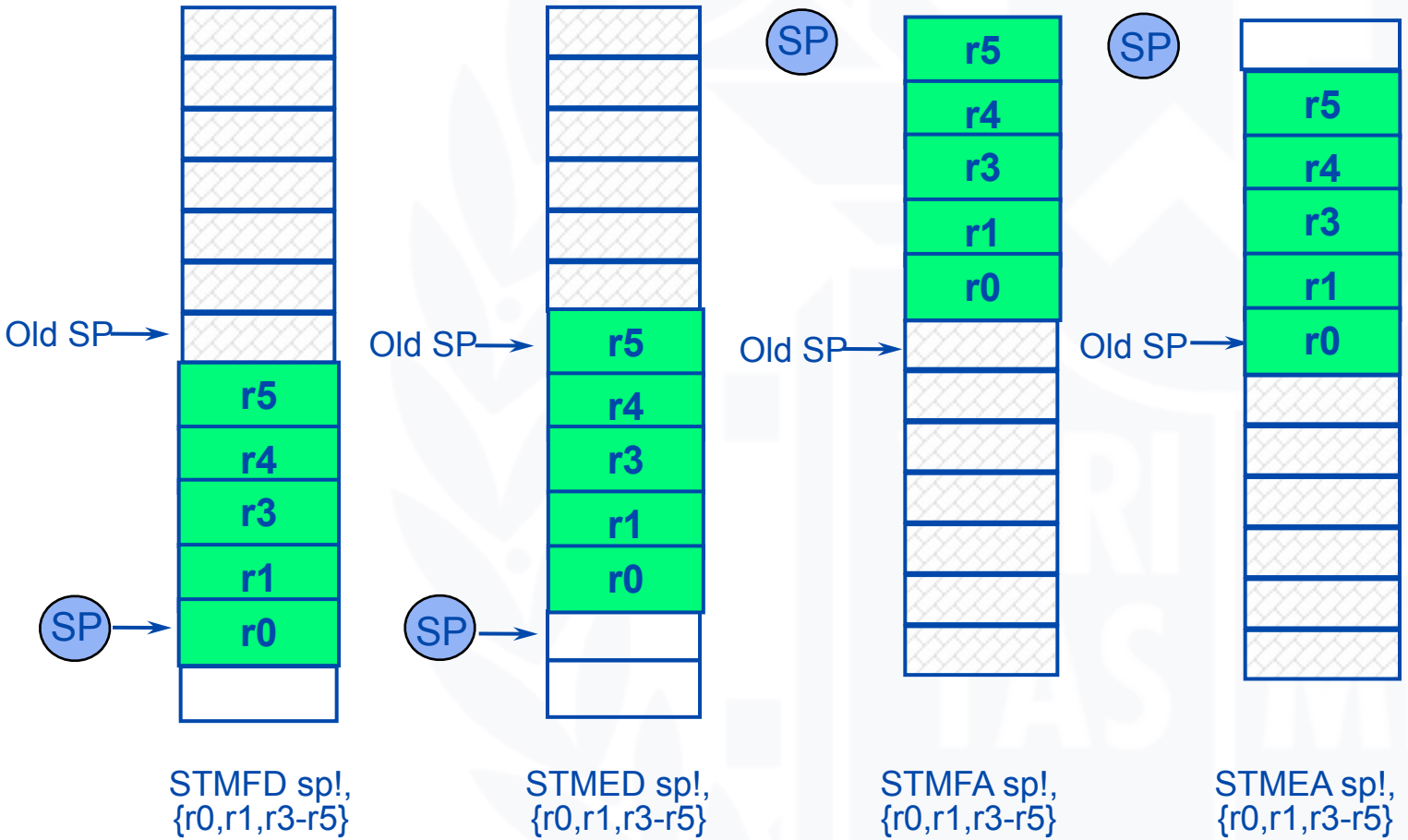


ELPL Embedded Low-Power Laboratory

Stack Operation

- Traditionally, a stack grows down in memory, with the last "pushed" value at the lowest address.
  - ARM also supports ascending stacks which grows up through memory.
- The value of the stack pointer can either:
  - Point to the last occupied address (Full stack) and so needs pre-decrementing (i.e. before the push)
  - Point to the next occupied address (Empty stack) and so needs post-decrementing (i.e. after the push)
- The stack type to be used is given by the postfix to the instruction:
  - STMFD / LDMFD : Full Descending stack
  - STMFA / LDMFA : Full Ascending stack.
  - STMED / LDMED : Empty Descending stack
  - STMEA / LDMEA : Empty Ascending stack
- Note: ARM Compiler will always use a Full descending stack.

Stack Examples



Stacks and Subroutines

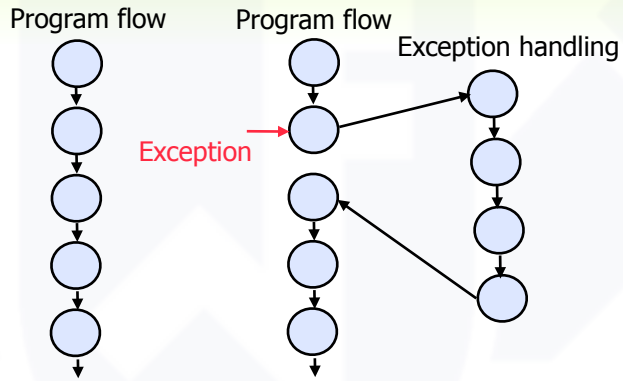
- One use of stacks is to create temporary register workspace for subroutines.
- Any registers that are needed can be pushed onto the stack at the start of the subroutine and popped off again at the end so as to restore them before returning to the caller :

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

- If the pop instruction also had the 'S' bit set (using '^') then the transfer of the PC when in a privileged mode would also cause the SPSR to be copied into the CPSR (see exception handling module).

Exception Handling and the Vector Table

- Exception
  - External interrupts
  - Divide by zero, overflow, etc.
  - Software interrupt
- Location of exception handling routines
  - Fixed location
    - Reset $00000000, NMI $00000004, etc.
  - Variable location with the vector table
    - Vector entry is fixed, e.g. $0 is reset, $4 is NMI, etc.
    - Jump to address values in the vector table: reset ($00000000)
    - If reset vector is set to $1000, Jump to $1000 when reset is asserted
- For safe return
  - Save all the previous contexts including registers
  - Save the program counter of the original next instruction
- Privilege mode
  - User mode program → exception → exception routine w/privilege mode → return to the user mode program w/user mode



Exception Handling and the Vector Table

- When an exception occurs,
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate CPSR bits
    - Enter ARM state if necessary
    - Mode field bits
    - Interrupt disable flags if appropriate.
  - Maps in appropriate banked registers
  - Stores the "return address" in LR_<mode>
  - Sets PC to vector address
  - To return, exception handler needs to:
    - Restore CPSR from SPSR_<mode>
    - Restore PC from LR_<mode>

Exception Handling and the Vector Table

Exception type	Exception mode	Vector address
Reset	Supervisor	0x00000000
Undefined instructions	Undefined	0x00000004
Software Interrupt (SWI)	Supervisor	0x00000008
Prefetch Abort (Instruction fetch memory abort)	Abort	0x0000000c
Data Abort (Data Access memory abort)	Abort	0x00000010
IRQ (Interrupt)	IRQ	0x00000018
FIQ (Fast Interrupt)	FIQ	0x0000001c

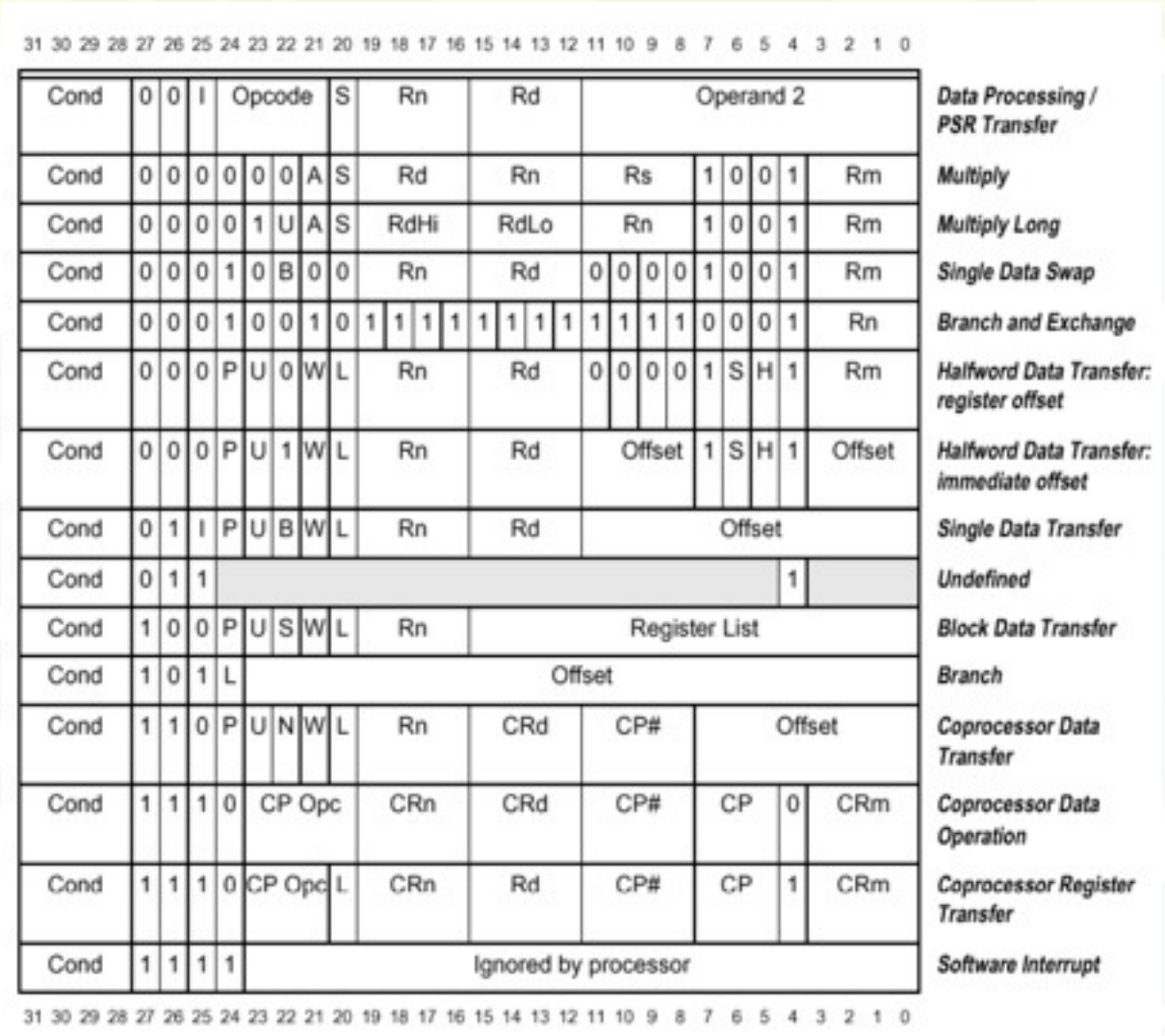
Exception Handling and the Vector Table

- Exception priority

Priority	Exception
1 (Higher)	Reset
2	Data abort
3	FIQ
4	IRQ
5 (Lowest)	Undefined instruction Software interrupt

ARM Instruction Set Format

- Instruction word length is 32-bits
- 36 instruction formats



Conditional Execution

- Branches to be executed conditionally
- Using the condition evaluation hardware, ARM effectively increases number of instructions
  - All instructions contain a condition field which determines whether the CPU will execute them
  - Non-executed instructions soak up 1 cycle
    - Still have to complete cycle so as to allow fetching and decoding of following instructions
- Removes the need for many branches, which stall the pipeline (3 cycles to refill)
  - Allows very dense in-line code, without branches
  - The time penalty of not executing several conditional instructions is frequently less than overhead of the branch or subroutine call that would otherwise be needed

Data Processing Instructions

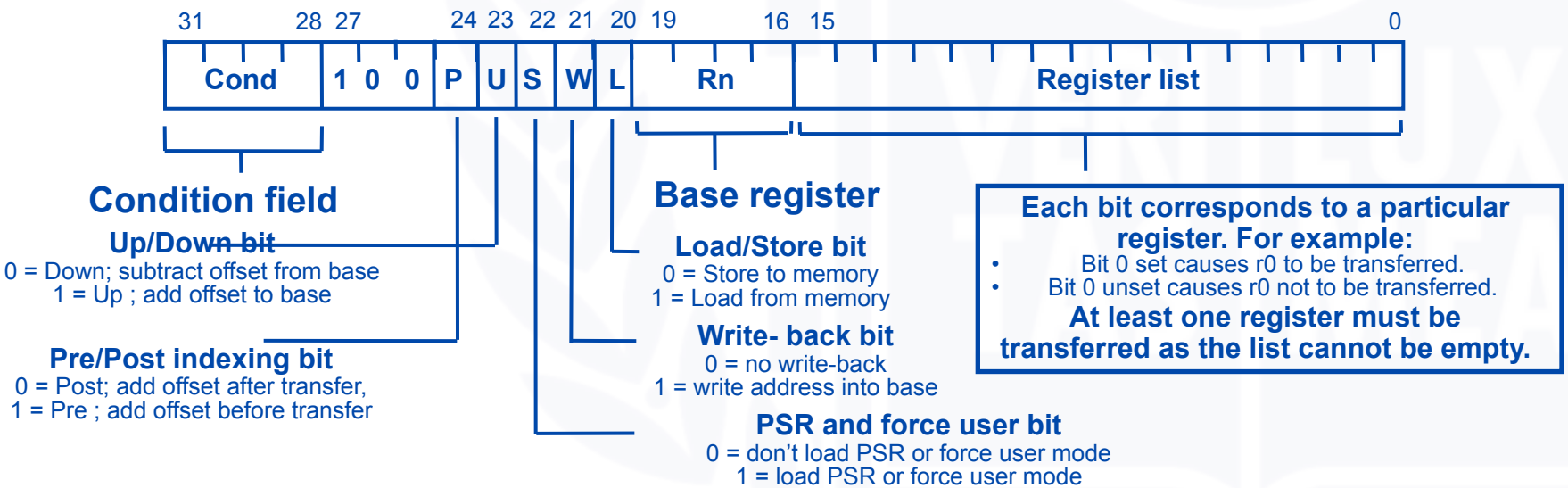
- All sharing the same instruction format.
- Contains:
  - Arithmetic operations
  - Comparisons (no results - just set condition codes)
  - Logical operations
  - Data movement between registers
- ARM is a load/store (register) architecture
  - These instructions only work on registers and NOT on memory.
- Perform a specific operation on one or two operands.
  - First operand always a register - Rn
  - Second operand sent to the ALU via barrel shifter.

Load/Store Instructions

- The ARM is a Load / Store Architecture:
  - Does not support memory to memory data processing operations.
  - Must move data values into registers before using them.
- This might sound inefficient, but in practice it isn't:
  - Load data values from memory into registers.
  - Process data in registers using a number of data processing instructions which are not slowed down by memory access.
  - Store results from registers out to memory.
- The ARM has three sets of instructions which interact with main memory. These are:
  - Single register data transfer (LDR / STR).
  - Block data transfer (LDM/STM).
  - Single Data Swap (SWP).

Block Data Transfer (1)

- The Load and Store Multiple instructions (LDM / STM) allow between 1 and 16 registers to be transferred to or from memory.
- The transferred registers can be either:
  - Any subset of the current bank of registers (default).
  - Any subset of the user mode bank of registers when in a privileged mode (postfix instruction with a '^').



Block Data Transfer (2)

- Base register used to determine where memory access should occur.
  - 4 different addressing modes allow increment and decrement inclusive or exclusive of the base register location.
  - Base register can be optionally updated following the transfer (by appending it with an '!').
  - Lowest register number is always transferred to/from lowest memory location accessed.

- These instructions are very efficient for
  - Saving and restoring context
    - Useful to view memory as a stack.
  - Moving large blocks of data around memory
    - Useful to directly represent functionality of the instructions.

Direct functionality of Block Data Transfer

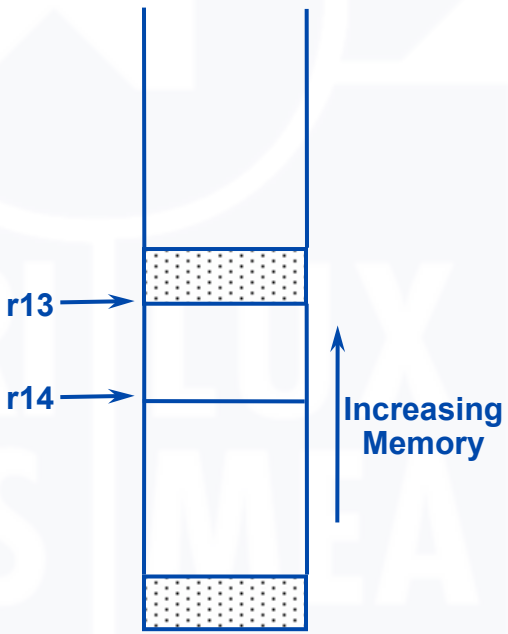
- When LDM / STM are not being used to implement stacks, it is clearer to specify exactly what the functionality of the instruction is:
  - i.e. specify whether to increment / decrement the base pointer, before or after the memory access.

- In order to do this, LDM / STM support a further syntax in addition to the stack one:
  - STMIA / LDMIA : Increment After
  - STMIB / LDMIB : Increment Before
  - STMDA / LDMDA : Decrement After
  - STMDB / LDMDB : Decrement Before

Example: Block Copy

- Copy a block of memory, which is an exact multiple of 12 words long, from the location pointed to by r12 to the location pointed to by r13. r14 points to the end of block to be copied.

```
; 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
```



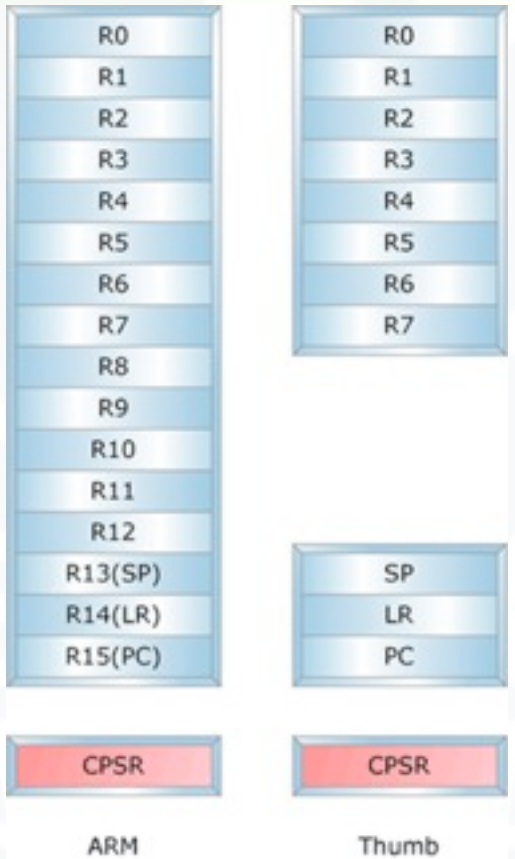
- This loop transfers 48 bytes in 31 cycles
- Over 50 Mbytes/sec at 33 MHz

Software Interrupt (SWI)

- In effect, a SWI is a user-defined instruction.
- It causes
  - an exception trap to the SWI hardware vector
  - a change to supervisor mode,
  - the associated state saving), and the SWI exception handler to be called.
- The handler can then examine the comment field of the instruction to decide what operation has been requested.
- By making use of the SWI mechanism, an operating system can implement a set of privileged operations, which, applications running in user mode can request.

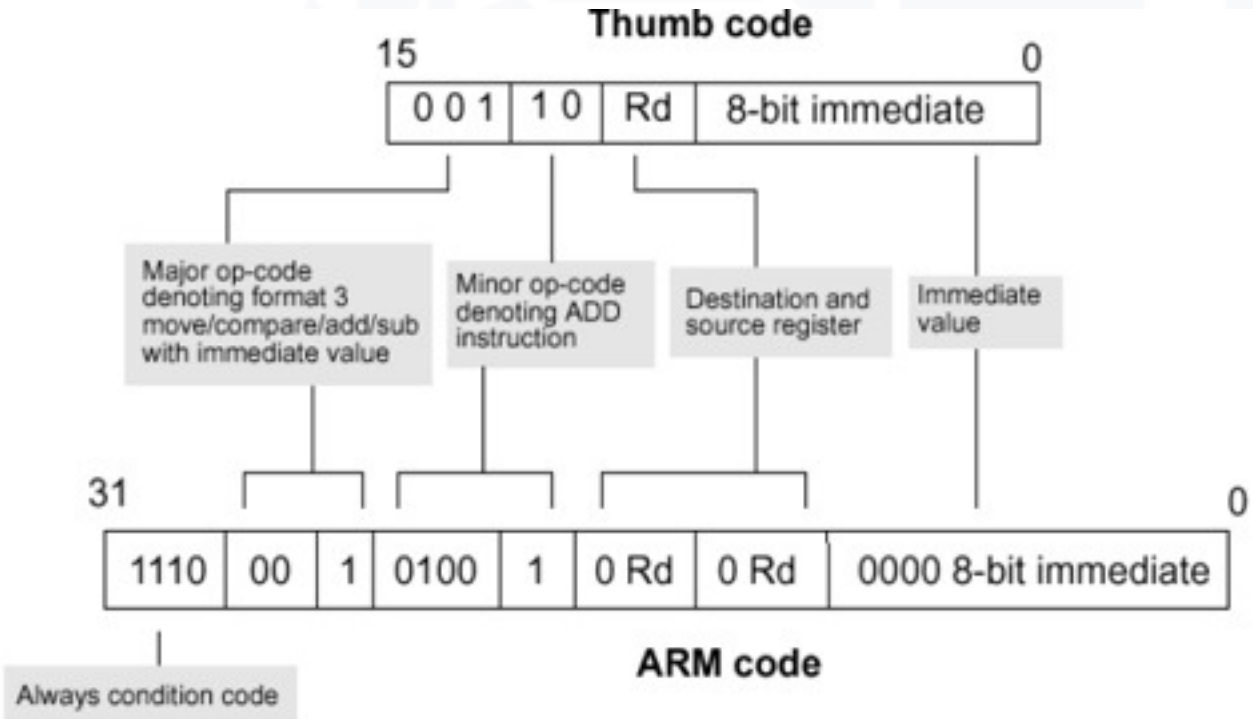
Thumb instructions

- T (Thumb)-extension shrinks the ARM instruction set to 16-bit word length
  - 35-40% saving in amount of memory compared to 32-bit instruction set
- Extension enables simpler and significantly cheaper realization of processor system. Instructions take only half of memory than with 32-bit instruction set without significant decrease in performance or increase in code size.
- Extension is made to instruction decoder at the processor pipeline
- Registers are preserved as 32-bit but only half of them are used



Thumb instructions

- ARM and Thumb instruction formats



Thumb instructions

- Instruction word length shrunk to 16-bits
- Instructions follow their own syntax but each instruction has it's native ARM instruction counterpart
- Due to shrinking some functionality is lost
- 19 different Thumb instruction formats