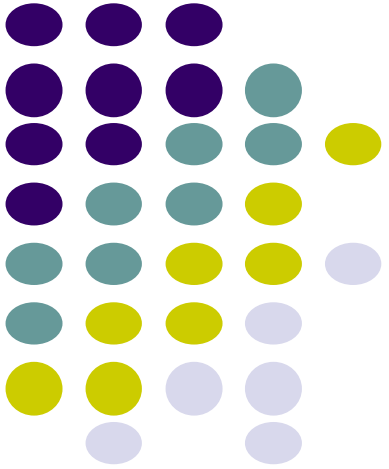
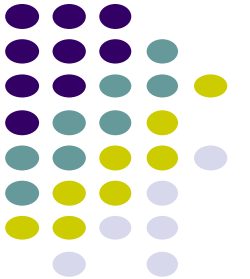
Understanding ARM Instruction Set Architecture (ISA) using ARM Simulator

Dr. Hari T.S. Narayanan

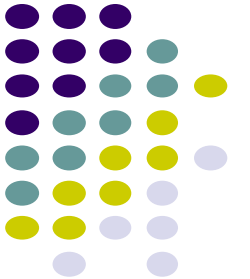


Pre-Requisite



- High School Computer Science
  - Arithmetic and Logic Unit (ALU)
  - Memory, Registers
  - C Programming
  - Compiler
  - **Assembly Level Language**
  - **Assembler**
  - Machine Code

Importance of Instruction Set



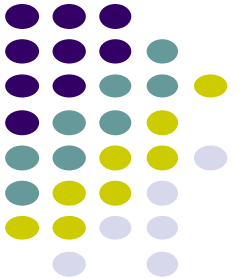
- Computer Organization course presents the design and development of Computing components (Memory, ALU, I/O Ports, Bus, etc)
- Computer Architecture course on the other hand
  - Defines the interfaces between various components within a Processor
  - Defines programmers' interface to a Processor using Instruction Set Architecture (ISA)

The ARM Instruction Set Architecture

Most of this material is taken from ARM University

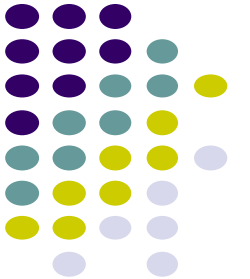
Why ARM ISA in Computer Organization Course; that too at the start?

Advanced RISC Machine (ARM)



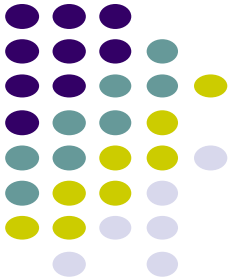
- ARM family includes 8/16/32/64 bits processors
- Different versions of ARM processors share the same machine Instruction Set
- We will be studying 32-bit ARM processor using a ARM Simulator application, **ARMSim#**
- ARMSim# © R. N. Horspool, W. D. Lyons, M. Serra Department of Computer Science, University of Victoria – Why ARMSIM#?

Main features of the ARM Instruction Set



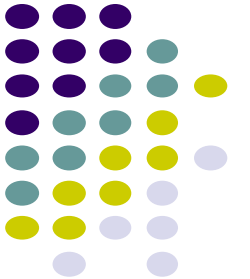
- All instructions are 32 bits long.
- ARM processor supports 8/16/32/64 bits data values
- ARM processor executes instructions in a **3-Stage pipeline**.
- Most of the ARM instructions executed in a **single cycle**.
- Every instruction can be **conditionally executed.**
- ARM uses **load/store architecture** (Register-Register Architecture)

Interrupt, Interrupt Handling, and Polling



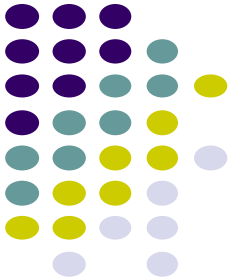
- In program in execution can be **Interrupted** by an I/O device or a Timer
- CPU suspends the current program*, handles the Interrupting I/O
- Once the I/O is completed CPU continues with the suspended program
- The interrupt handling itself can be Interrupted by another higher priority I/O device
- **Interrupt Handler** is the piece of code that process the I/O or timer
- Interrupt is an efficient alternative to **Polling** or **Busy waiting**
- **Interrupt is not the only thing that can alter the flow of execution of a program**

Processor Modes



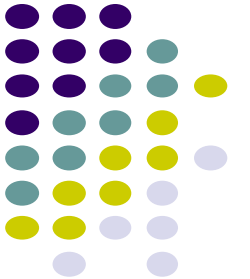
- The ARM has six **operating modes:**
  - User (unprivileged mode under which most tasks run)
  - **FIQ** (entered when a high priority (**fast**) **interrupt** occurs)
  - **IRQ** (entered when a low priority (**normal**) **interrupt** occurs)
  - **Supervisor** (entered on reset and when a **Software Interrupt instruction** is executed)
  - **Abort** (used to handle memory access violations)
  - **Undef** (used to handle undefined instructions)

Addressing



- ARM includes capability to address byte, half word, and full word (32 bits)
- ARM **instructions are always 32 bits in size**, data can be of 8/16/32 bits
- This means, addresses of instructions start from **0 and increase in steps of 4**
- Instructions are loaded from **Word** boundary – **Instruction Offset**
- **Default Instruction offset is 0**
- **Program Counter** is initialized to 0 when the processor is reset
- The **least** significant 2 -bits of Program Counter are always 0
- This reduces the number of bits required to store **address and address offset of instructions**

Registers

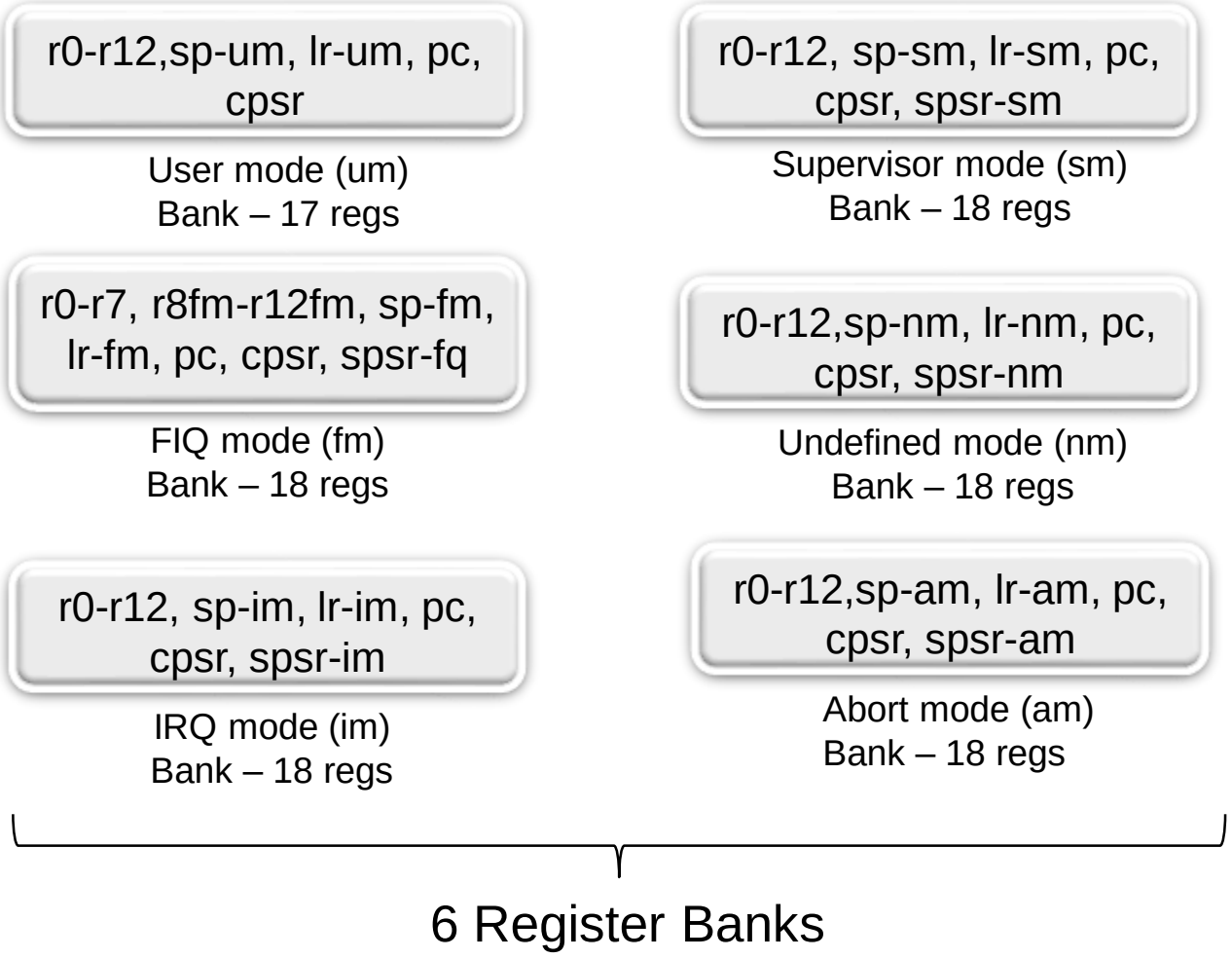


- **Registers** are high speed and low capacity (8/16/32, …) memory
- ALU operates on data in Data Registers –r0, r1,…., r12
- There are other special purpose Registers; each with specific role
- Examples: **Program Counter (PC); Process Status Register (PSR)**
- **Register File and Register banks** are physical and logical grouping of Registers respectively
- ARM supports 37 Registers in all

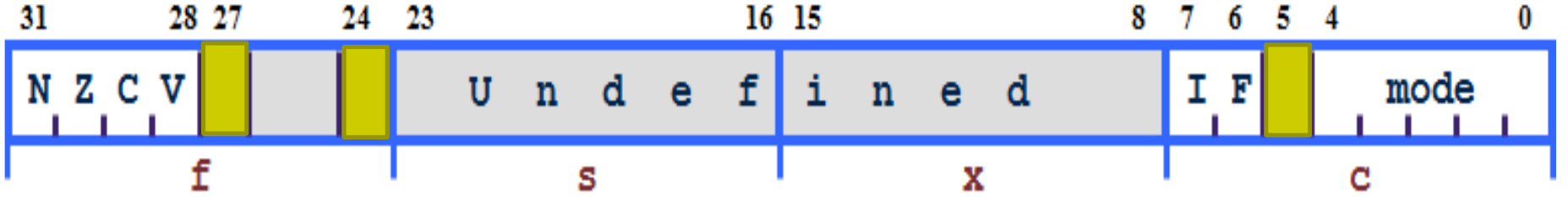
Register Banks

37 Registers

1. 18 General Purpose Registers: r0 – r12, r8fm-r12fm (fm stands of FIQ Mode)
2. 1 Program Counter: pc
3. 6 Stack Pointer (sp): sp-xx
4. 6 Link Register (lr): lr-xx
5. 1 Current Program Status Register: cpsr
6. 5 Saved Program Status Register: spsr-xx



Program Status Registers (PSR)



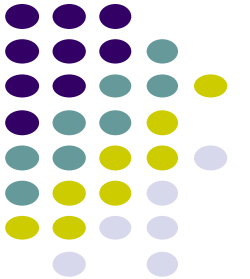
- Condition code flags
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation oVerflowed

Carry versus overflow!

- Interrupt Disable bits.
  - I = 1: Disables the IRQ.
  - F = 1: Disables the FIQ.
- Mode bits
  - Specify the processor mode

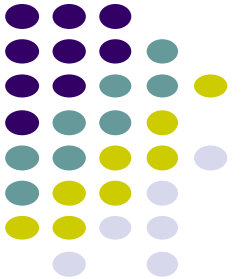
Mode Encoding	
Value	Mode
10000	User
10001	FIQ
10010	IRQ
10011	SVC
10100	Abort
10101	Undef
11111	System

Hands-on Exercises



- Copy ARM Simulator (**Armsim#**) folder
- Install ARM Simulator
- Check if the Simulator installed correctly
- Start the simulator
- Load a sample program (test.s)
- Explore the features of the Simulator

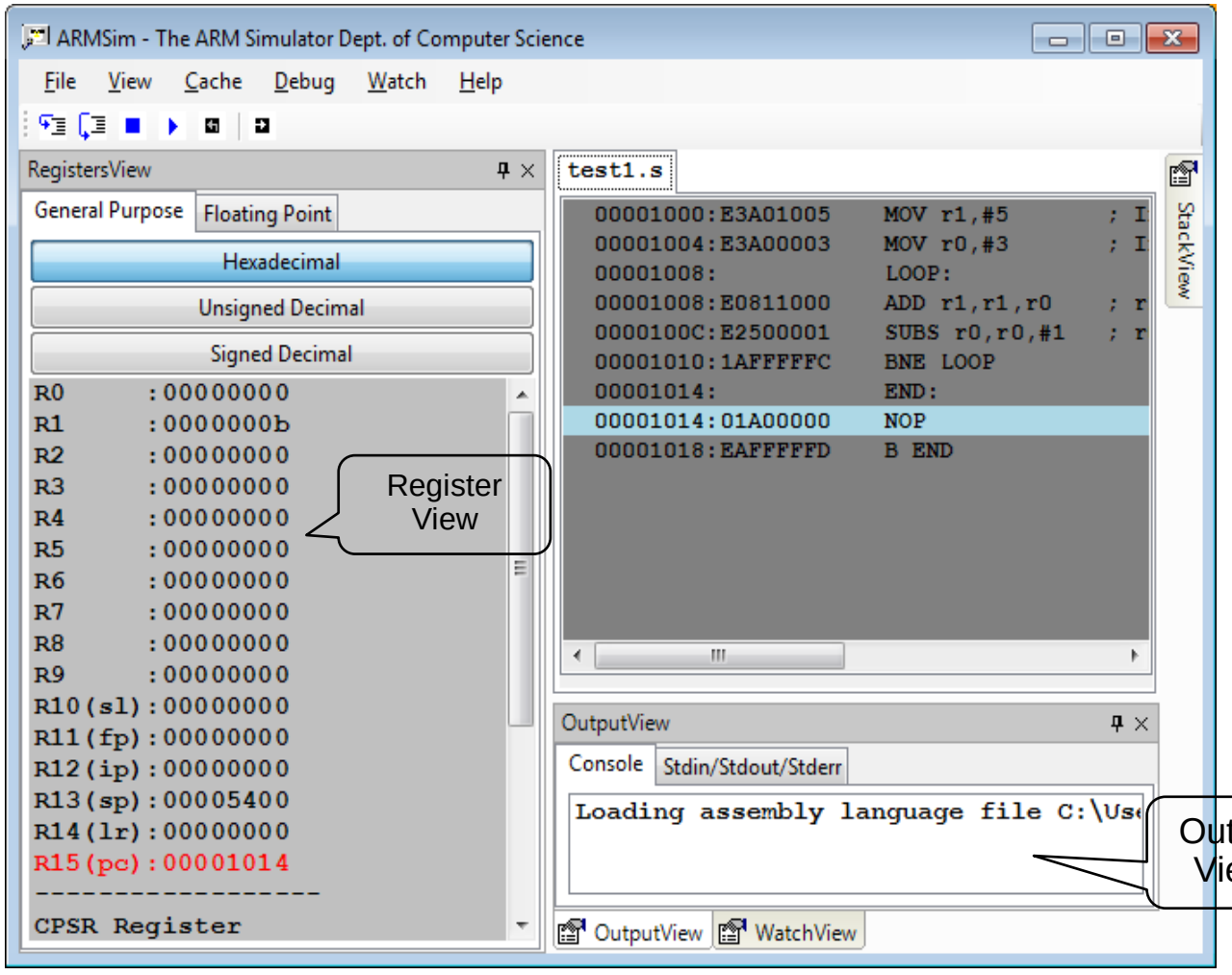
Ex 1: Installing ARMSim



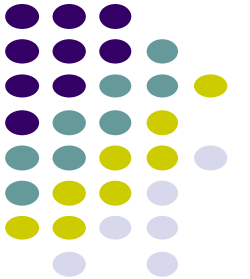
- Install ARMSim 1.91 from ARMSim folder that you copied from my public folder
- Note: This simulator (Windows, Linux and Mac versions available) can also be downloaded from: http://armsim.cs.uvic.ca/Downloads.html.
- Override your Anti-virus, if the download fails for some reason
- The file is about 1300 KB
- Follow the default install options. ARMSim installs in few seconds.
- The Simulator is installed in "C:/Program Files (x86)/University of Victoria"
- There will be short cut installed on your desktop
- Launch the simulator to validate your installation

Ex 2: Understanding ARM Simulator UI

- Once you successfully launch ARM Simulator
- Go to "File" menu and using "Load" option, load test1.s file
- Note: If the **test.s** is not loaded it will not be possible to view the register contents.
- Views
- Registers
- Instructions & Instruction Encoding
- Menu items (Note: Watch & Cache later)

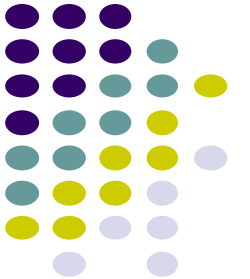


Ex 3: User Mode Bank Registers



- Observe the contents of r0 (R0) to r12 (R12)
- R13 is Stack Pointer (SP)
- R14 is Link Register (LR)
- R15 is Program Counter (PC)
- **Current Process Status Register** (**CPSR**) is shown with its bit map
- Correlate R15 (PC) content with Code View instruction address, and memory content in memory view

Exception Handling



- Exceptions are events that interrupt the flow of currently running program
- Exceptions are handled by exception specific code
- This **code is loaded (context Switched)** and given control after completing the current instruction.
- The **Context Switching** is realized by h/w (processor core)
- **Vector table** is an important data structure used in Context Switching
- Some Exceptions are: I/O and Timer Interrupts, Reset, SWI

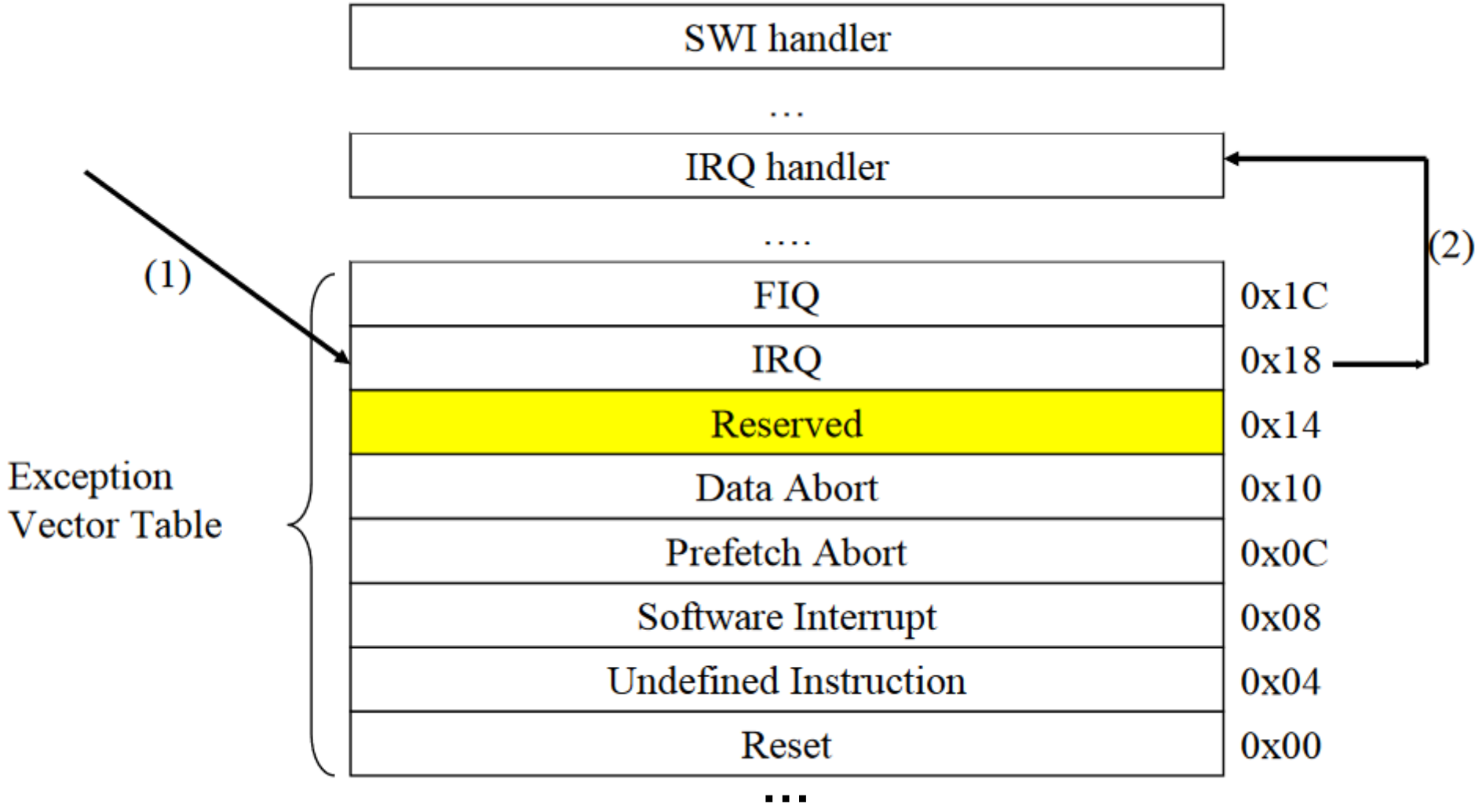
Exception Handling and the Vector Table

- When an exception occurs, **the core:**
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate CPSR bits (**especially the mode**)
  - Maps in appropriate banked registers
  - Stores the "*return address*" in LR_<mode>
  - Sets PC to vector address
    - For instance for Software Interrupt PC is set to 0x00000008
- To return, exception handler (sw) needs to:
  - Restore CPSR from SPSR_<mode>
  - **Restore PC from LR_<mode>**

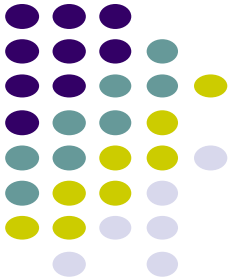
0x00000000	**Reset**	LDR	PC, Reset_Addr
0x00000004	**Undefined Instruction**	LDR	PC, Undefined_Addr
0x00000008	**Software Interrupt**	LDR	PC, SWI_Addr
0x0000000C	**Prefetch Abort**	LDR	PC, Prefetch_Addr
0x00000010	**Data Abort**	LDR	PC, Abort_Addr
0x00000014	**Reserved**	NOP	
0x00000018	**IRQ**	LDR	PC, IRQ_Addr
0x0000001C	**FIQ**	LDR	PC, FIQ_Addr

http://osnet.cs.nchu.edu.tw/powpoint/Embedded94_1/Chapter%207%20ARM%20Exceptions.pdf

Memory Map


SWI handler
…
IRQ handler
….
(1)
(2)
FIQ
IRQ
Reserved
Data Abort
Prefetch Abort
Software Interrupt
Undefined Instruction
Reset
0x1C
0x18
0x14
0x10
0x0C
0x08
0x04
0x00
Exception
Vector Table
…

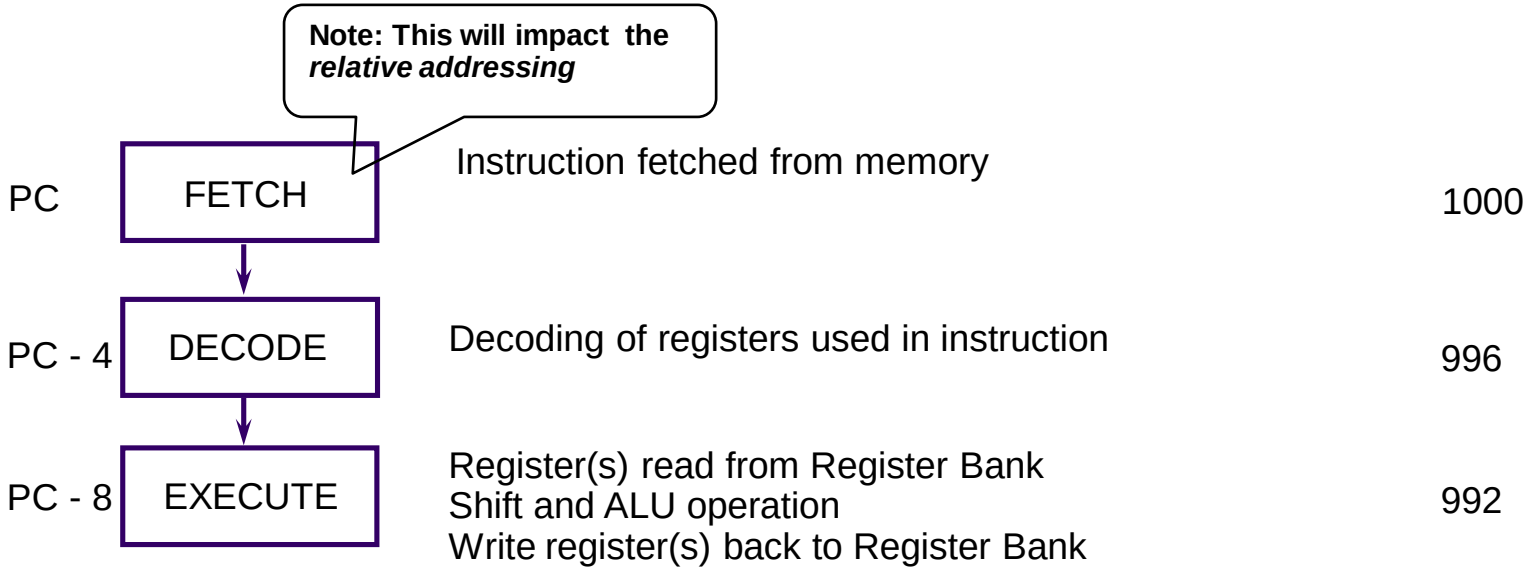
Branching to Absolute Address



- Load the PC with the absolute address of the instruction to be executed!
- This could be done only in certain **modes** – example SWI (Supervisory)
- Example:
  - ldr pc, = 0X3000

The Instruction Pipeline

- In order to increase the *instruction throughput* ARM uses a **3-stage pipeline**



- PC points to the instruction being fetched, rather than pointing to the instruction being executed.

Branching Instruction with Offset

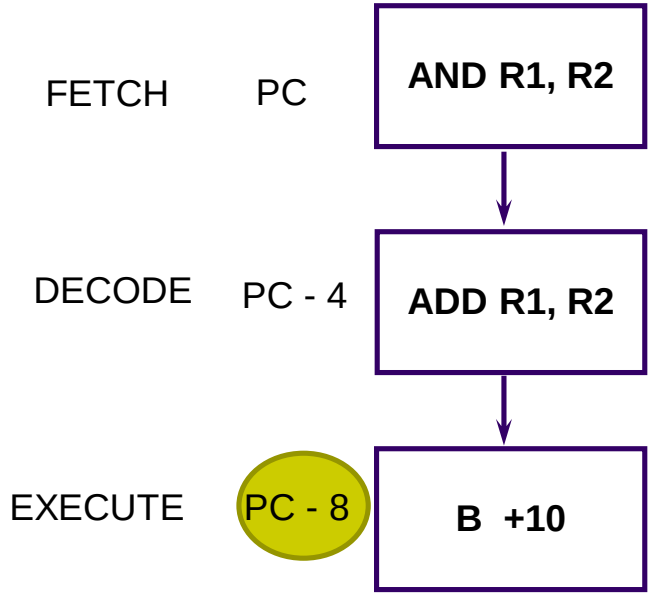
- The ARM supports B(ranch) instructions with **offset address** (rather than **absolute address**)
- The **target address** to branch is computed at execution using an **offset** (actual offset*4) & **Program Counter** (PC) Value:

  target address = PC + offset x 4;

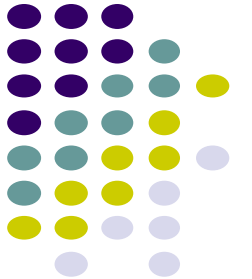
  = (**branch instruction address** + 8) + (**offset** << 2)
- Similarly the *offset address* can be computed from *target* and branch instruction address as follows:

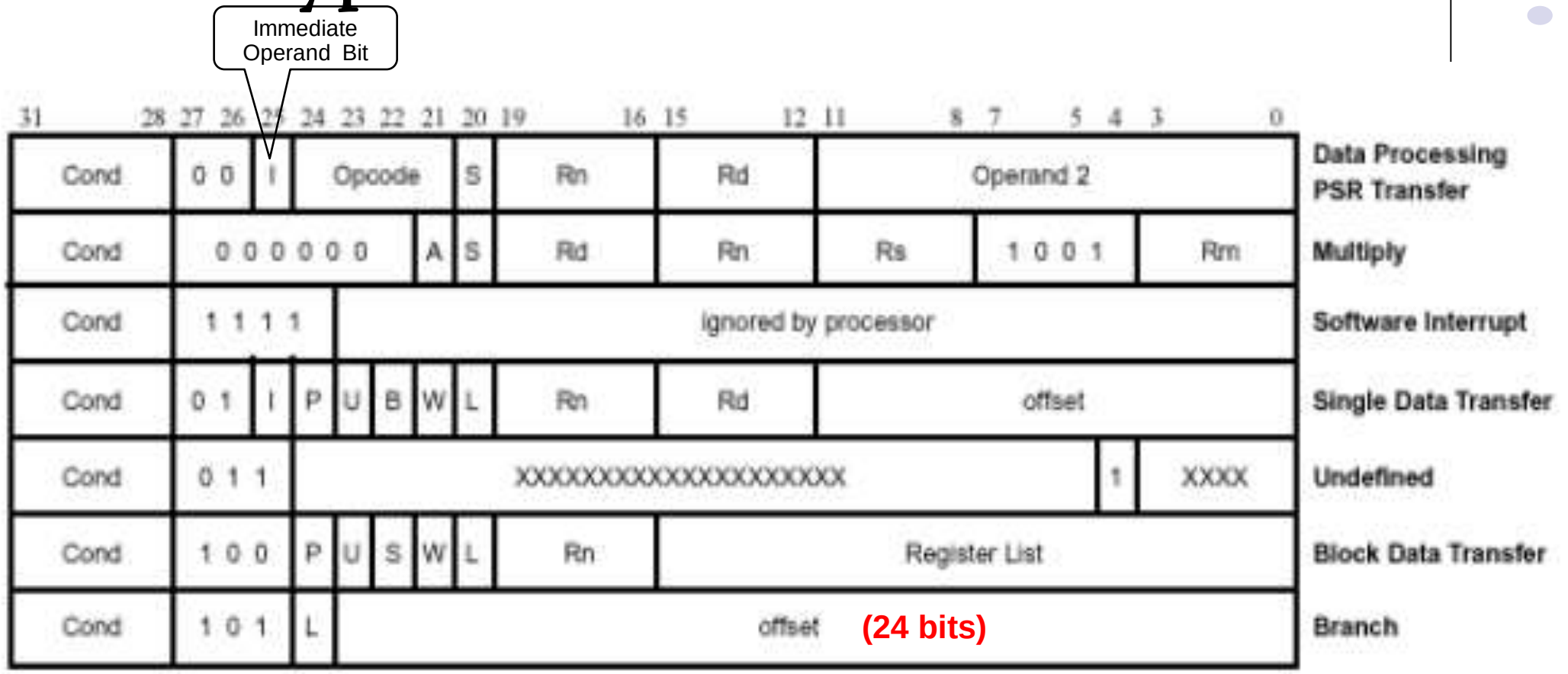
  Offset = (target address – PC) / 2

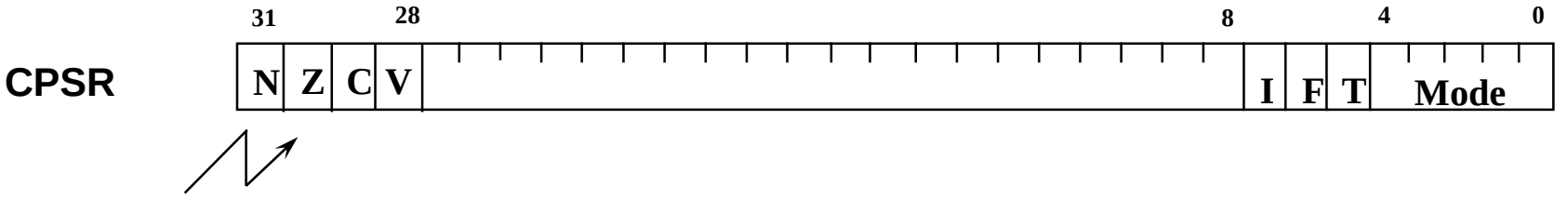
  = (**target address** - (**branch instruction address** +8)) >> 2



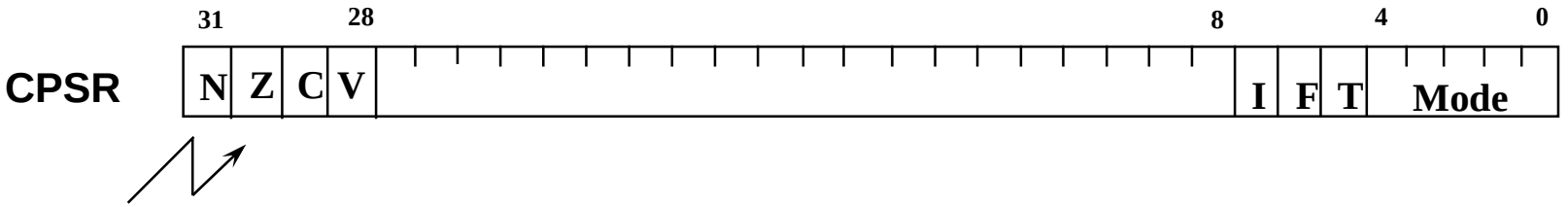
Instruction Encoding for Some ARM Instruction Types





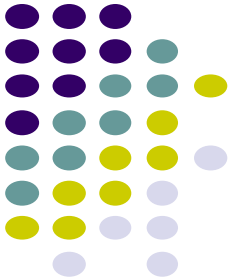


The CPSR and Condition Field



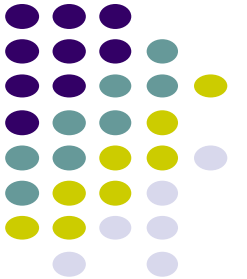
Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Conditional Execution



- Most processors allow only branches to be executed conditionally.
- However by reusing 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.
- This 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.

Using and updating the Condition Field



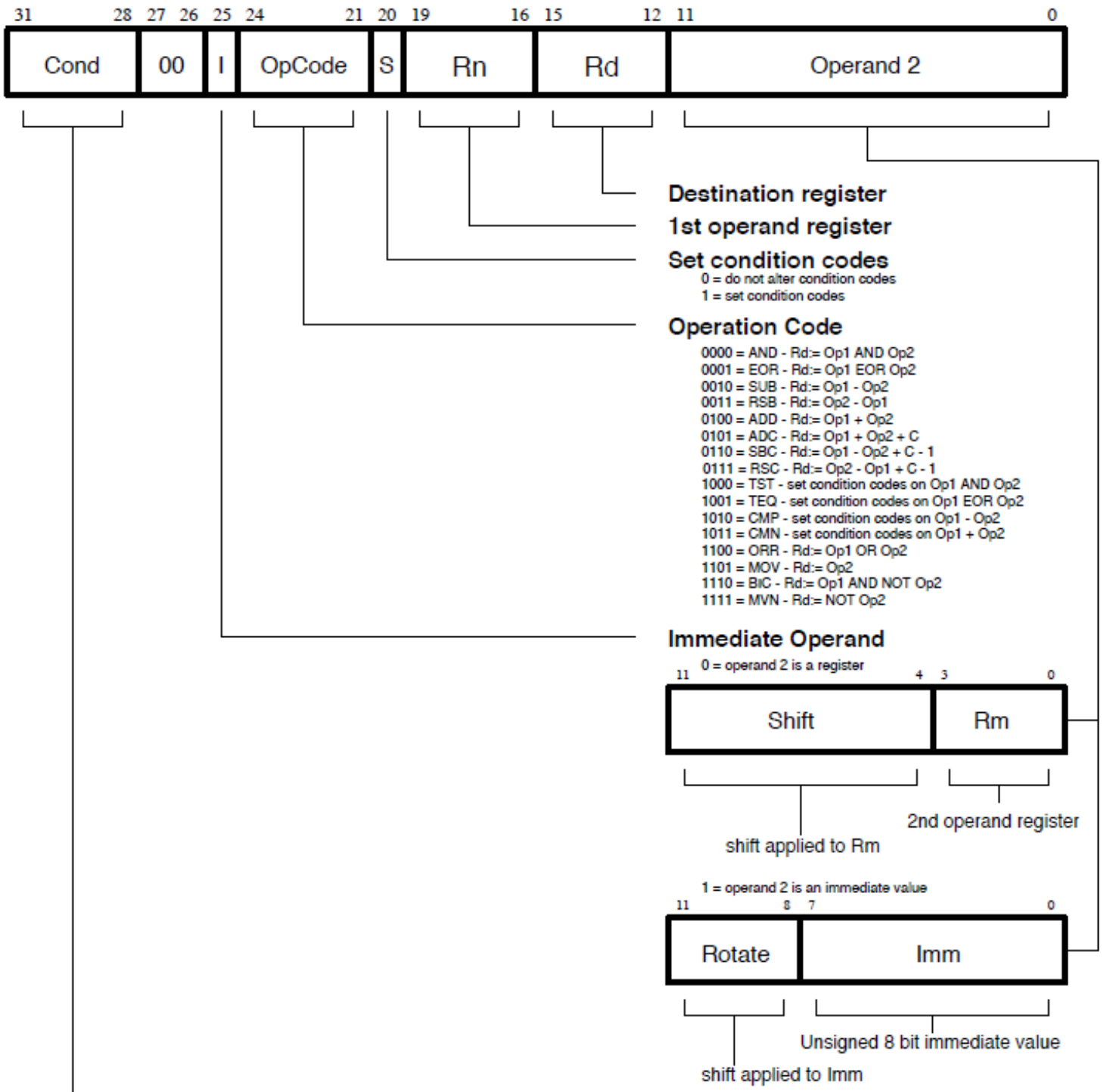
- To execute an instruction conditionally, simply postfix it with the appropriate condition:
  - For example an add instruction takes the form:
    - `ADD r0,r1,r2 ; r0 = r1 + r2 (ADDAL)`
  - To execute this only if the zero flag is set:
    - `ADDEQ r0,r1,r2 ; If zero flag set then r0 = r1 + r2`
- By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect).
- To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an "S".
  - For example to add two numbers and set the condition flags:
    - `ADDS r0,r1,r2 ;`
    - `r0 = r1 + r2 ; ... and set flags`

✓ Ex 4: PC, Address, Memor

- Add r1, r1, r0

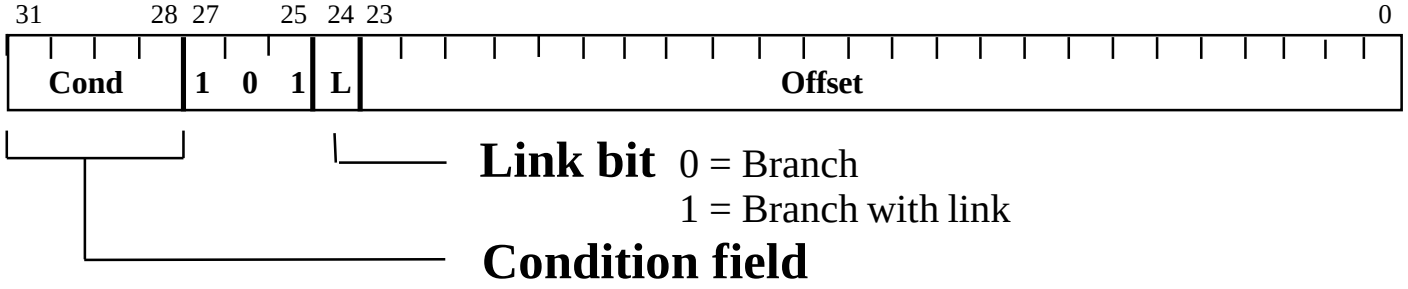
1110 00 0 0100 0 0001 0001 00000000 0000

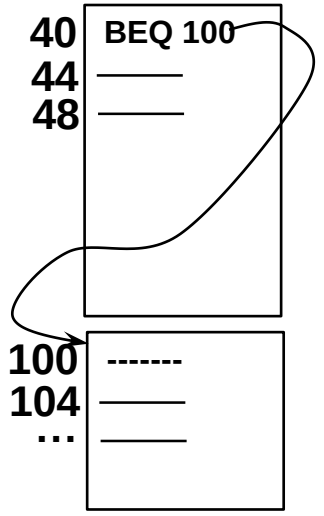
- Binary representation of the above Add instruction



Branch instruction

- `B{<cond>} label ;` versus Branch with Link





- The *offset* for branch instructions is calculated by the assembler:
  - By taking the difference between the target instruction and the branch instruction address minus 8 (to allow for the pipeline). Then right shift the result by 2 positions.
  - This gives a range of ± 32 Mbytes.

Ex 6: Branch Example (Forward Branch) ✓

- The *target address* to branch is computed as a function *offset address* with respect to *Program Counter (PC) Value*

  target address

  = PC + *offset*4*

  = (***branch instruction address*** + 8) + (***offset*** << 2)

- Similarly the *offset address* can be computed from *target* and *branch instruction addresses* as follows:

  offset

  = (*target address* – PC) >> 2

  = (***target address*** - (***branch instruction address*** +8)) >> 2

- **Branch Instruction address: 0x1010; Target Address: 0x1024**

  Offset = ?; From offset compute Target Address

  Now using this offset and Branch instruction address find target address

- Note: use test2.s

test2.s

```
00001000:E3A01005    MOV r1,#5
00001004:E3A00003    MOV r0,#3
00001008:E3A02008    mov r2,#8
0000100C:            LOOP:
0000100C:E0821000    ADD r1,r2,r0
00001010:E2500001    SUBS r0,r0,#1
00001014:0A000003    BEQ END
00001018:EAFFFFFB    B LOOP
0000101C:01A00000    NOP
00001020:01A00000    NOP
00001024:01A00000    NOP
00001028:            END:
00001028:01A00000    NOP
0000102C:01A00000    NOP
00001030:01A00000    NOP
00001034:EAFFFFFB    B END
```

Test2.s

Ex 7: Branch Example (Backward Branch) ✓

- The *target address* to branch is computed as a function *offset address* with respect to *Program Counter (PC) Value*

  target address

  = PC + (*offset address*)

  = (*branch instruction address* + 8) + (*offset* << 2)

- Similarly the *offset address* can be computed from *target* and *branch instruction addresses* as follows:

  offset

  = (*target address* – PC) << 2

  = (*target address* - (*branch instruction address* +8)) >> 2

- Branch Instruction address: 0x1014; Target Address: 0x1008

  Offset = ?; From offset compute Target Address

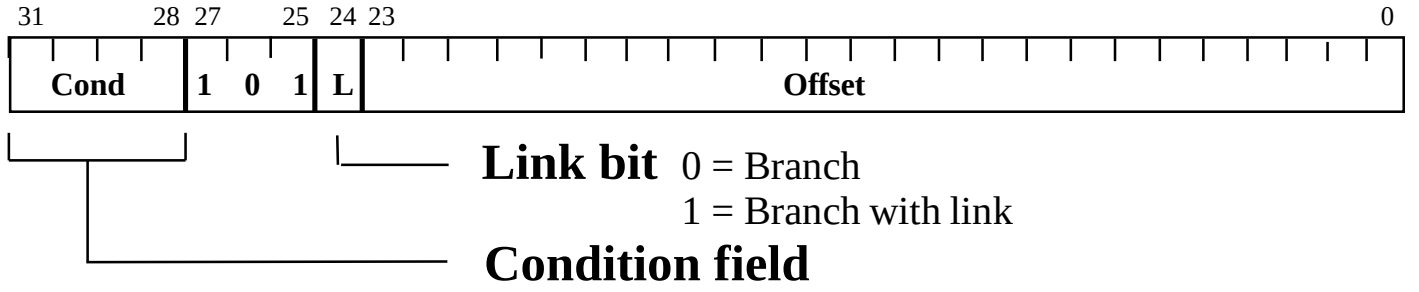
  Now using this offset and Branch instruction address find target address

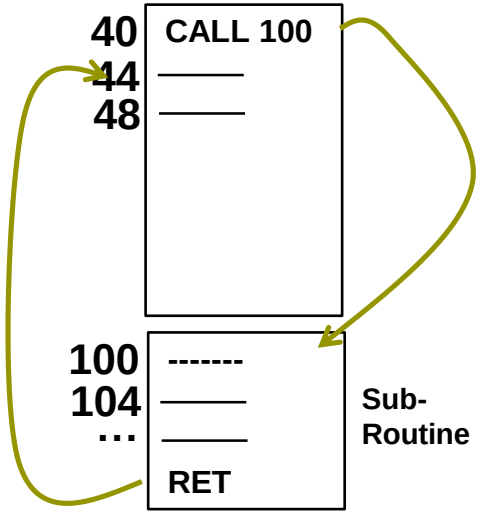
test2.s

```
00001000:E3A01005    MOV r1,#5
00001004:E3A00003    MOV r0,#3
00001008:E3A02008    mov r2,#8
0000100C:            LOOP:
0000100C:E0821000    ADD r1,r2,r0
00001010:E2500001    SUBS r0,r0,#1
00001014:0A000003    BEQ END
00001018:EAFFFFFB    B LOOP
0000101C:01A00000    NOP
00001020:01A00000    NOP
00001024:01A00000    NOP
00001028:            END:
00001028:01A00000    NOP
0000102C:01A00000    NOP
00001030:01A00000    NOP
00001034:EAFFFFFB    B END
```

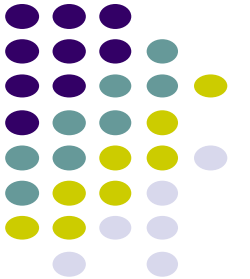
Branch instruction with Link

- Branch with Link : `BL{<cond>} sub_routine_label`



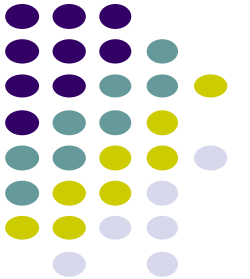


Data processing Instructions



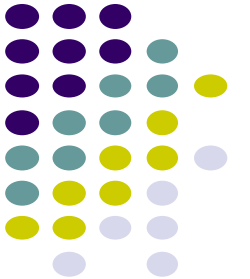
- Largest family of ARM instructions, all sharing the same instruction format.
- Contains:
  - Arithmetic operations
  - Comparisons (no results - just set condition codes)
  - Logical operations
  - Data movement between registers
- Remember, this is a load / store architecture
  - These instruction only work on registers, *NOT* memory.
- They each perform a specific operation on one or two operands.
  - **First operand always a register - Rn**
  - **Second operand sent to the ALU via barrel shifter.**
- We will examine the barrel shifter shortly.

Arithmetic Operations



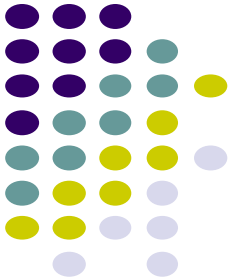
- Operations are:
  - ADD operand1 + operand2 ; operand1 is Rn
  - ADC operand1 + operand2 + carry
  - SUB operand1 - operand2
  - SBC operand1 - operand2 + carry -1
  - RSB operand2 - operand1
  - RSC operand2 - operand1 + carry - 1
- Syntax:
  - <Operation>{<cond>}{S} Rd, Rn, Operand2 ; Rd is destination register
- Examples
  - ADD r0, r1, r2
  - SUBGT r3, r3, #1
  - RSBLES r4, r5, #5

Comparisons



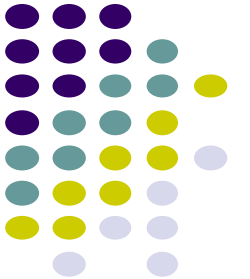
- The only effect of the comparisons is to
  - ***UPDATE THE CONDITION FLAGS***. Thus no need to set S bit.
- Operations are:
  - CMP operand1 - operand2, but result not written
  - CMN operand1 + operand2, but result not written
  - TST operand1 AND operand2, but result not written; **Bitwise AND**
  - TEQ operand1 EOR operand2, but result not written; **Bitwise EXOR**
- Syntax:
  - <Operation>{<cond>} Rn, Operand2
- Examples:
  - CMP r0, r1
  - TSTEQ r2, #5

Logical Operations



- Operations are:
  - AND operand1 AND operand2
  - EOR operand1 EOR operand2
  - ORR operand1 OR operand2
  - BIC operand1 AND NOT operand2 [Bit Clear when operand2 = opearand1]
- Syntax:
  - <Operation>{<cond>}{S} Rd, Rn, Operand2 ; Rd ← Rn op Operand2
- Examples:
  - AND r0, r1, r2
  - BICEQ r2, r3, #7
  - EORS r1,r3, r0

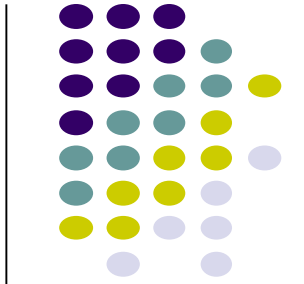
Data Movement



- Operations are:
  - MOV Rd, operand2
  - MVN Rd, NOT operand2

  Note that these instruction make no use of operandI.
- Syntax:
  - <Operation>{<cond>}{S} Rd, Operand2
- Examples:
  - MOV r0, rI
  - MOVS r2, #10
  - MVNEQ rI, #0

The Barrel Shifter



- The ARM doesn't have actual shift instructions.
- Instead it has a barrel shifter which provides a mechanism to carry out shifts as part of other **instructions with second operand**
- Barrel shifter performs shift operations faster than regular shifter
- So what operations does the barrel shifter support?

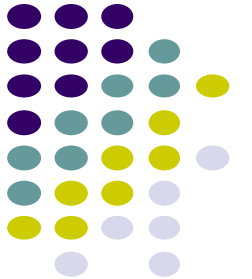
Barrel Shifter - Left Shift

- Shifts left by the specified amount (multiplies by powers of two) e.g.

  LSL #5 = multiply by 32

- CF is Carry Flag bit

Barrel Shifter - Right Shifts



Shift Right - Logical

Shifts right by the specified unsigned amount (divides by powers of two) e.g.

LSR #2 = divide by 4

	7	6	5	4	3	2	1	0	Comment
	0	0	1	0	0	0	0	0	Representing 32
									Do LSR 1
0 →	0	0	0	1	0	0	0	0	This is 16
									Do LSR 1
0 →	0	0	0	0	1	0	0	0	This is 8

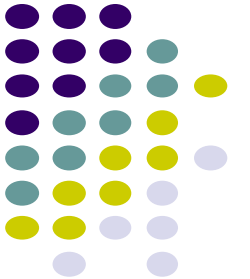
Shift Right - Arithmetic

Shifts right (divides by powers of two) and preserves the sign bit, for 2's complement operations. e.g.

ASR #2 = divide by 4

7	6	5	4	3	2	1	0	Comment
1	1	1	0	0	0	0	0	Representing -32 in 2s complement
								Do ASR 1
1	1	1	1	0	0	0	0	This is -16 in 2s complement form
								Do ASR 1
1	1	1	1	1	0	0	0	This is -8 in 2s complement form

Barrel Shifter - Rotations

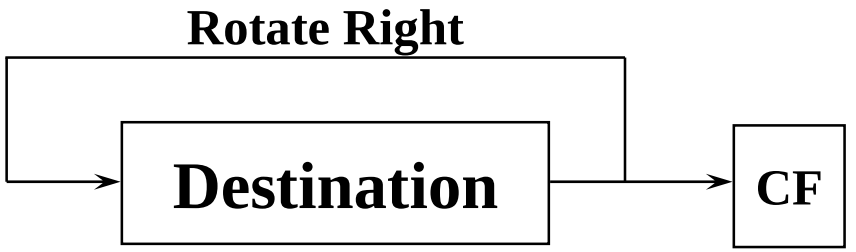


Rotate Right (ROR)

- Similar to an ASR but the bits wrap around as they leave the LSB and appear as the MSB.

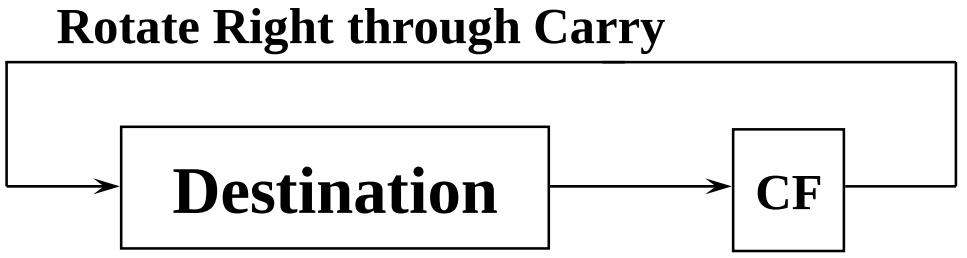
  e.g. ROR #4

- Note the last bit rotated is also used as the Carry Out.

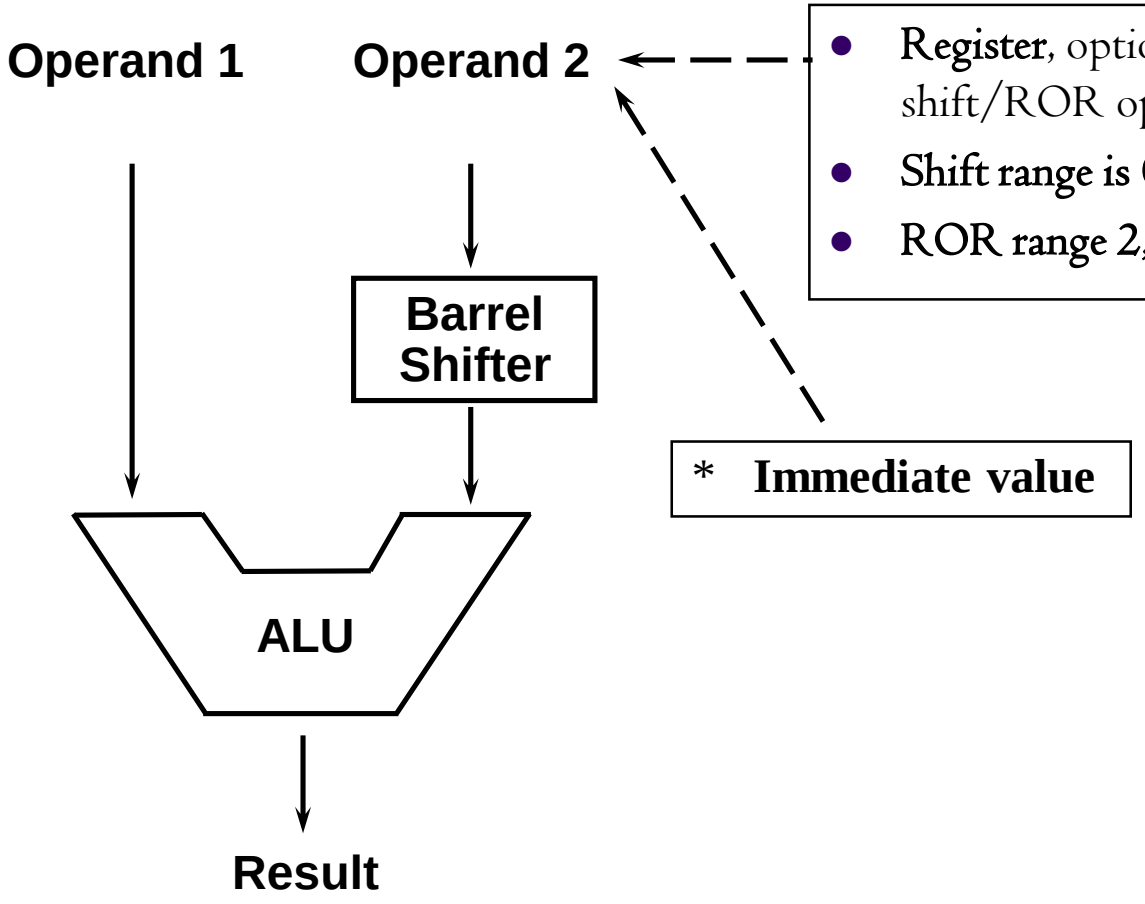


Rotate Right Extended (RRX)

- This operation uses the CPSR C flag as a 33rd bit.
- Rotates right by 1 bit. Encoded as ROR #0.



Using the Barrel Shifter: The Second Operand

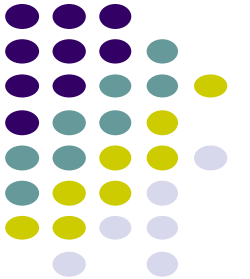


- **Register**, optionally with shift/ROR operation applied.
- **Shift range is 0 to 31**
- **ROR range 2,4,6, …, 30**

Examples

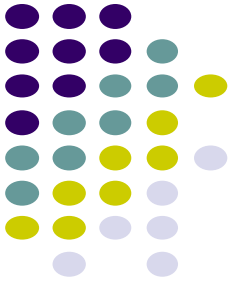
- `MOV r0, r0, LSL #1`
  - Multiply R0 by two.
- `MOV r1, r1, LSR #2`
  - Divide R1 by four (unsigned).
- `MOV r2, r2, ASR #2`
  - Divide R2 by four (signed).
- `MOV r3, r3, ROR #16`
  - Swap the top and bottom halves of R3.
- `ADD r4, r4, r4, LSL #4`
  - Multiply R4 by 17. (N = N + N * 16)
- `RSB r5, r5, r5, LSL #5`
  - Multiply R5 by 31. (N = N * 32 – N)

Ex - Using Shifted Register ✓



- Using a multiplication instruction to multiply by a constant means first loading the constant into a register and then waiting a number of internal cycles for the instruction to complete.
- A more optimum solution can often be found by using some combination of MOVs, ADDs, SUBs and RSBs with shifts.
  - Multiplications by a constant equal to a ((power of 2) ± 1) can be done in one cycle.
- Example: r0 = r1 * 5 ; MUL R1, R1, #5
  = r1 + (r1 * 4)
  - ADD r0, r1, r1, LSL #2
- Example: r2 = r3 * 105
  = r3 * 15 * 7
  = r3 * (16 - 1) * (8 - 1)
  - RSB r2, r3, r3, LSL #4 ; r2 = r3 * 15
  - RSB r2, r2, r2, LSL #3 ; r2 = r2 * 7

Large constants with limited size of Immediate Value



- Immediate operand size is 12 bits.
- Out of this, the least significant 8 bits are used for constant value
- Rest of the 4 bits are used of ROR
- Value 1 in these bits code ROL 2
- Value 2 in these bits code ROL 4
- …
- Value 15 in these bits code ROL 30

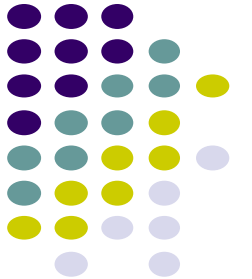
Second Operand: Immediate Value (2)

- This gives us:
  - 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]
- These can be loaded using, for example:
  - MOV r0, #0x40, 26 ; => MOV r0, #0x1000 (ie 4096)
- To make this easier, the assembler will convert to this form for us if simply given the required constant:
  - MOV r0, #4096 ; => MOV r0, #0x1000 (ie 0x40 ror 26)
- The bitwise complements can also be formed using MVN:
  - MOV r0, #0xFFFFFFFF ; assembles to MVN r0, #0
- If the required constant cannot be generated, an error will be reported.

Notes ✓

- ROR 30 is same ROL (32-30)
- ROR 0 gives us constants 0 – 255 (0000 0000 to IIII IIII)
- ROR 30 or ROL 2 gives constants 256, 260, … I020 (0I00 0000 to 0xFF)
- ROR 28 or ROL 4 gives constants I024, I040, … , 4080
- ROR 26 or ROL 6 gives constants 4096, 4I28, … , I0208
- And so on … till ROR 2

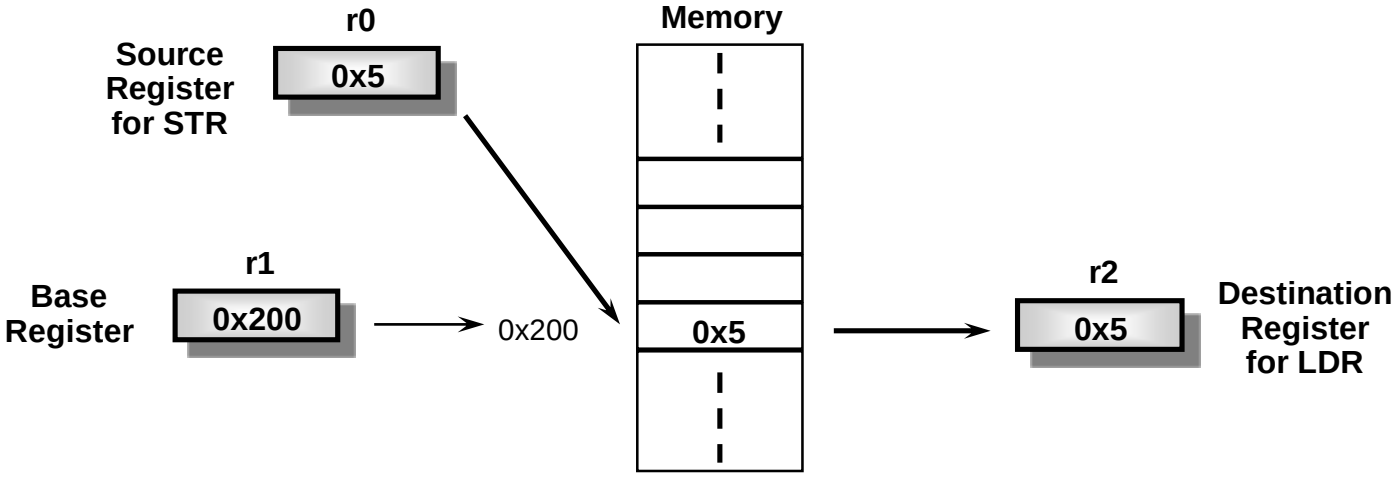
Load / Store Instructions



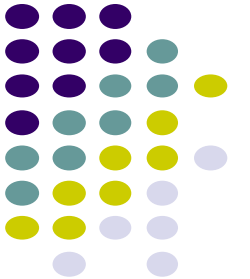
- The ARM is a Load / Store Architecture:
  - Does not support memory to memory data processing operations.
  - Must move data values into registers before using them.
- This might sound inefficient, but in practice isn't:
  - Load data values from memory into registers.
  - Process data in registers 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).

Load and Store

- The memory location to be accessed is held in a base register
  - STR r0, [r1] ; Store contents of r0 to location pointed to ; by contents of r1.
  - LDR r2, [r1] ; Load r2 with contents of memory location ; pointed to by contents of r1.



Load/Store variations & Syntax

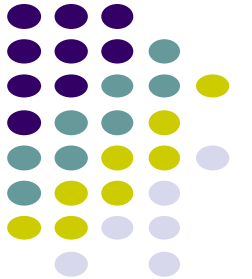


- The basic load and store instructions are:
  - Load and Store Word or Byte
    - LDR / STR / LDRB / STRB
- ARM Architecture Version 4 also adds support for half words and signed data.
  - Load and Store Half word
    - LDRH / STRH
  - Load Signed Byte or Half word - load value and sign extend it to 32 bits.
    - LDRSB / LDRSH
- All of these instructions can be conditionally executed by inserting the appropriate condition code after STR / LDR.
  - e.g. LDREQB
- Syntax:
  - <LDR|STR>{<cond>}{<size>} Rd, <address>

Exercise

- Write ARM code snippet that store some data in 5 consecutive half words starting from memory address 0x2000 and retrieve those 5 half words and print the content on stdout

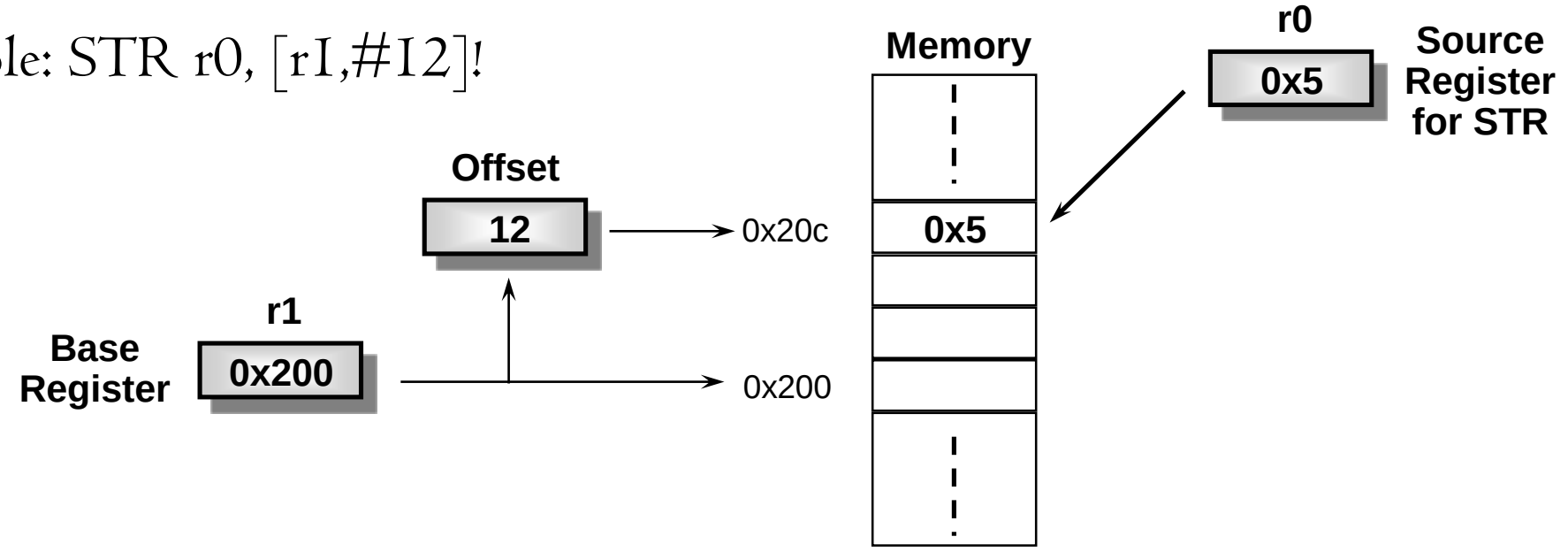
Load and Store with Indexing



- As well as accessing the actual location contained in the base register, these instructions can access a location offset from the base register pointer.
- This **offset** can be
  - An unsigned **12bit immediate value** (ie 0 - 4095 bytes).
  - A register, optionally shifted by an immediate value
- This can be either **added or subtracted** from the base register:
  - Prefix the offset value or register with '**+**' (default) or '**-**'.
- This **offset** can be applied:
  - before the transfer is made: ***Pre-indexed addressing***
    - optionally *auto-incrementing* the base register, by postfixing the instruction with an '**!**'.
  - after the transfer is made: ***Post-indexed addressing***
    - causing the base register to be *auto-incremented*.

Load and Store Word Pre-indexed Addressing

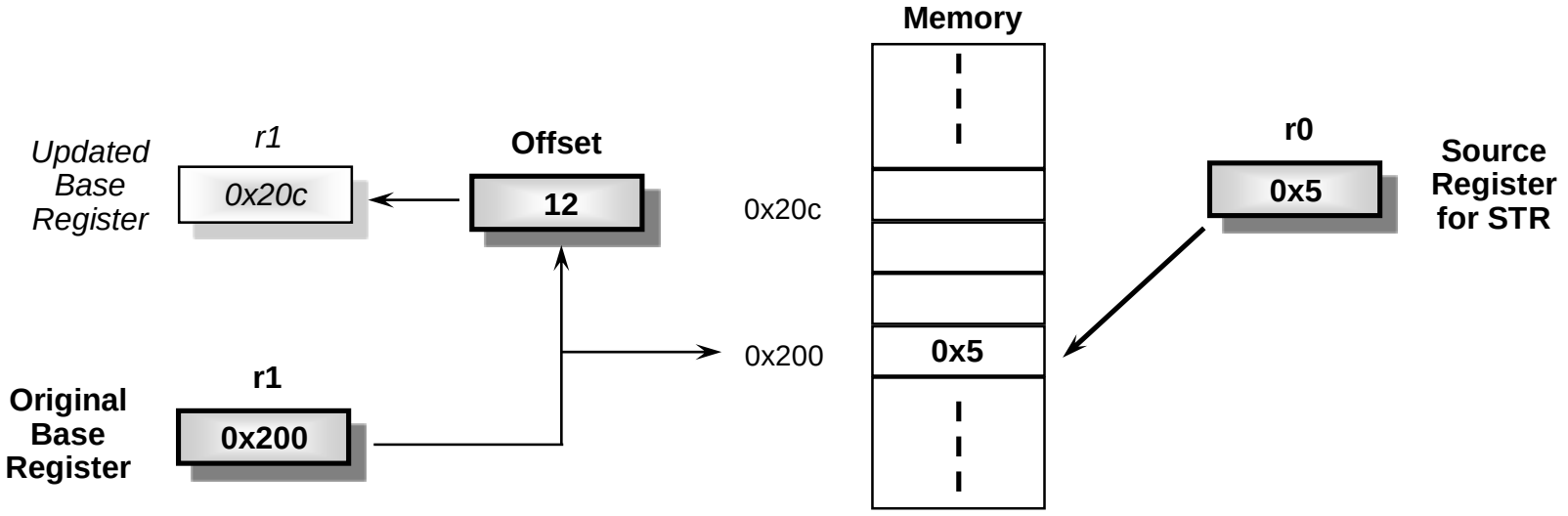
- Example: STR r0, [r1,#12]!



- To auto-pre- increment base pointer to 0x20c use: STR r0, [r1, #12]!
- To auto-pre-decrement base pointer to 0x1f4 use: STR r0, [r1,#-12]!
- Immediate operand can be a shifted Register!

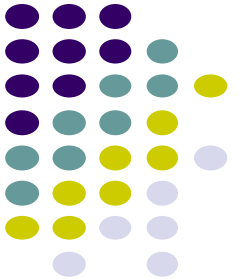
Load and Store Word Post-indexed Addressing

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



- To auto-post-increment the base register to location 0x1f4 instead use:
  - STR r0, [r1], #-12

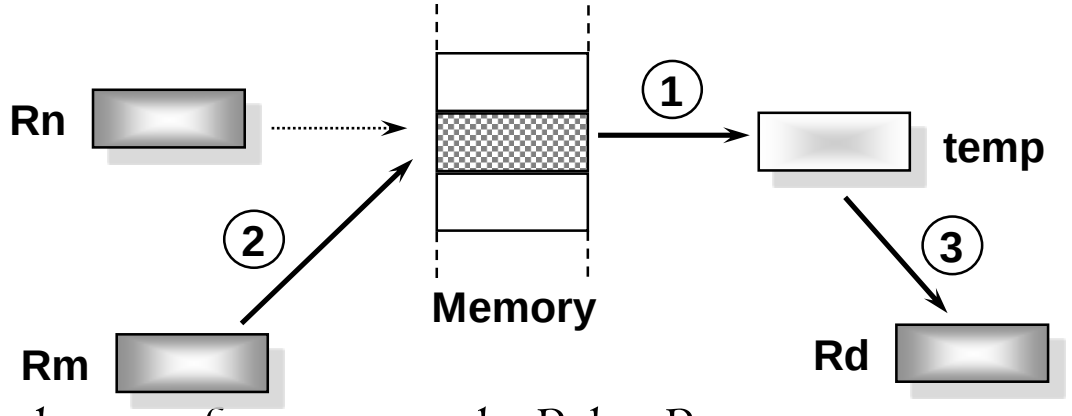
Code Snippets Pre-Indexing and Post-Indexing



- Develop ARM assembly code snippet that does Pre-indexing and Post-indexing. Demonstrate this with ARM Simulator
- Use Data Cache for various Block size to show how mis-rate decreases with larger block size
- Also justify the hit and miss count for instruction cache

Swap Instructions

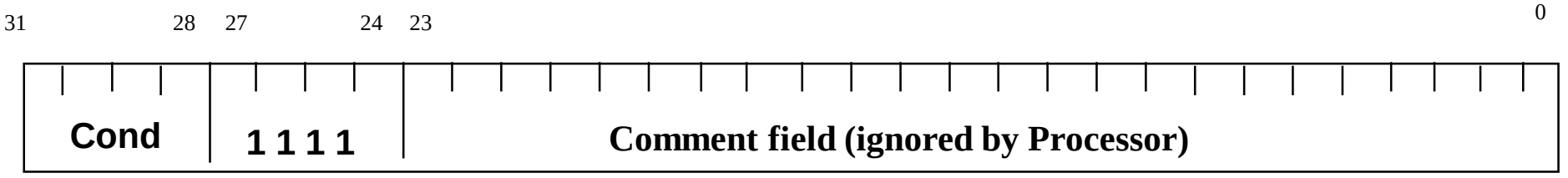
- Atomic operation of a memory read followed by a memory write which moves byte or word quantities between registers and memory.
- Syntax:
  - SWP{<cond>}{B} Rd, Rm, [Rn]



- Thus to implement an actual swap of contents make Rd = Rm.
- The compiler cannot produce this instruction.

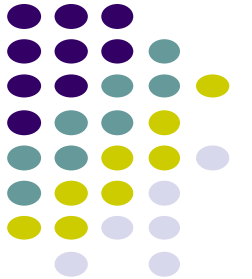
Interrupting an Instruction

Software Interrupt (SWI)

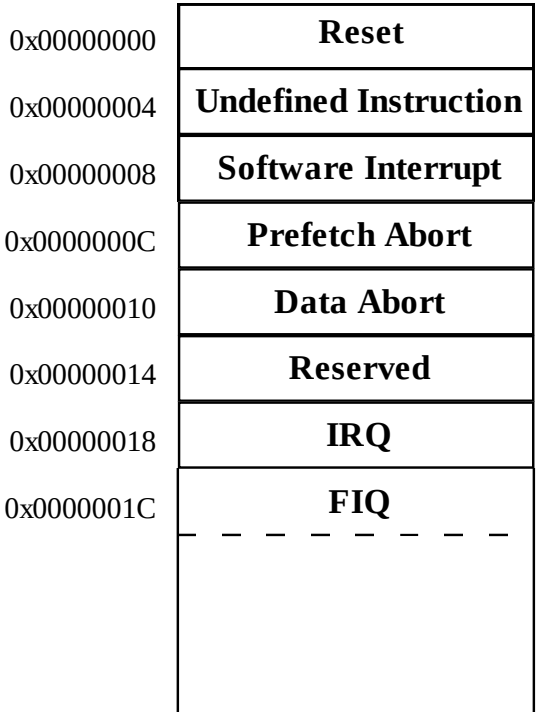


- In effect, a SWI is a user-defined instruction.
- It causes an exception trap to the SWI hardware vector (thus causing a change to supervisor mode, plus the associated state saving), thus causing the SWI exception handler to be called.
- The handler can then examine the comment field of the instruction 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.
- See Exception Handling Module for further details.

Exception Handling and the Vector Table



- When an exception occurs, the core:
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate CPSR bits
  - Maps in appropriate banked registers
  - Stores the "*return address*" in LR_<mode>
  - Sets PC to vector address
    - For instance for *Undefined instruction* PC is set to 0x00000008
- To return, exception handler needs to:
  - Restore CPSR from SPSR_<mode>
  - Restore PC from LR_<mode>



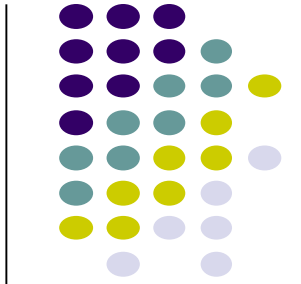
Assembler Directives

- Hello: **.asciz** "Hello Class\n“
- **.equ** SEG_A, 0x80
- **.equ** Ten, 10
- **.set** counter, 1
- **.set** counter, counter+1
- Address: **.word 0**
- **.align**

Reference:

https://community.arm.com/processors/b/blog/posts/useful-assembler-directives-and-macros-for-the-gnu-assembler

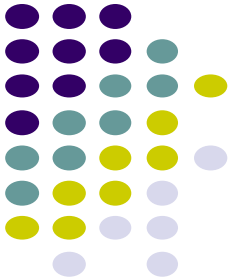
SWI – ARMSIM Specific I/O



- There are two plugins with ARMSIM
  1. Basic I/O Plugin
     - File I/O
     - Memory management
  2. Embest Board Plugin
     1. LED
     2. Keypad
     3. Buttons
     4. LED Number Segment

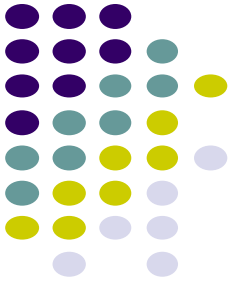
The I/O with SWI is unique to ARMSim – this is NOT a standard practice

Printing a Character & a string to STDOUT ✓



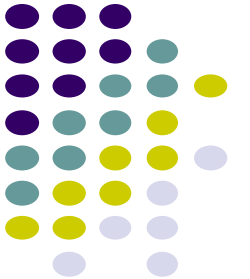
```
Hello: .asciz   "Hello Class!\n"
       mov      r0, #'I'
       swi      0x00
       mov      r0,#':'
       swi      0x00
       ldr      r0, =Hello
       swi      0x02
End:
       b End
```

SWI Comment Field Encoding for ARMSim



Opcode	Description and Action	Inputs	Outputs	EQU
`swi 0x00`	Display Character on Stdout	r0: the character		`SWI_PrChr`
`swi 0x02`	Display String on Stdout	r0: address of a null terminated ASCII string	(see also 0x69 below)	
`swi 0x11`	Halt Execution			`SWI_Exit`
`swi 0x12`	Allocate Block of Memory on Heap	r0: block size in bytes	r0:address of block	`SWI_MeAlloc`
`swi 0x13`	Deallocate All Heap Blocks			`SWI_DAlloc`
`swi 0x66`	Open File (mode values in r1 are: 0 for input, 1 for output, 2 for appending)	r0: file name, i.e. address of a null terminated ASCII string containing the name r1: mode	r0:file handle If the file does not open, a result of -1 is returned	`SWI_Open`
`swi 0x68`	Close File	r0: file handle		`SWI_Close`
`swi 0x69`	Write String to a File or to Stdout	r0: file handleor Stdout r1: address of a null terminated ASCII string		`SWI_PrStr`

SWI Comment Field Encoding for ARMSim



Opcode	Description and Action	Inputs	Outputs	EQU
swi 0x6a	Read String from a File	r0: file handle r1: destination address r2: max bytes to store	r0: number of bytes stored	SWI_RdStr
swi 0x6b	Write Integer to a File	r0: file handle r1: integer		SWI_PrInt
swi 0x6c	Read Integer from a File	r0: file handle	r0: the integer	SWI_RdInt
swi 0x6d	Get the current time (ticks)		r0: the number of ticks (milliseconds)	SWI_Timer

File Open & Close

```
InFileName:   .asciz   "Infile1.txt"
InFileError:  .asciz   "Unable to open input file\n"
     .align
InFileHandle:.word     0
```

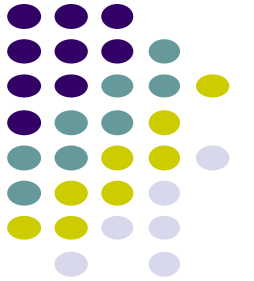
The following lines of code open the file called "Infile1.txt" for input and store its file "handle", returned in R0 by the opening call, into the appropriate memory location:.

```
    ldr  r0,=InFileName        @ set Name for input file
    mov  r1,#0                 @ mode is input
    swi  SWI_Open              @ open file for input
    bcs  InFileError           @ if error?
    ldr  r1,=InFileHandle      @ load input file handle
    str  r0,[r1]               @ save the file handle
```

File Close

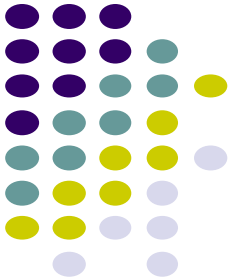
```
load the file handle
ldr       r0,=InFileHandle
ldr       r0,[r0]
swi       SWI_Close
```

File Input



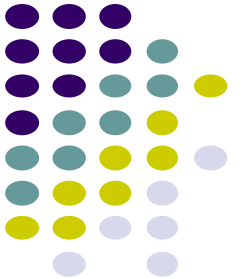
```
        ldr  r0,=InFileHandle
        ldr  r0,[r0]
        ldr  r1,=CharArray
        mov  r2,#80
        swi  0x6a
        bcs  ReadError
        ...
InFileHandle: .word 0
CharArray: .skip 80
```

Exercise



- Read Section 9 of ARMSim User Guide with objective to write code for **Embest Board Plugin**
- **This section will not be taught in the class – you are on your own!**
- **A similar exercise on Cache performance measurement with ARMSIM will be given in four weeks now.**

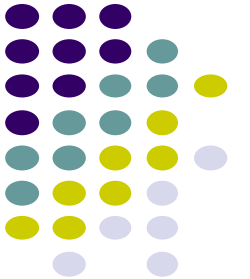
Multiplication Instruction



- ARM implement Booth's algorithm for multiplication
- **Long** and **Accumulate** options are supported
- Syntax for non-long multiplication:

- **Multiply**
  - MUL{<cond>}{S} Rd, Rm, Rs ; Rd = Rm * Rs
- **Multiply Accumulate** - does addition for free
  - MLA{<cond>}{S} Rd, Rm, Rs,Rn ; Rd = (Rm * Rs) + Rn
- **Restrictions on use:**
  - Rd and Rm cannot be the same register
    - Can be avoided by swapping Rm and Rs around. This works because multiplication is commutative.

Syntax for Long Multiplication ✓

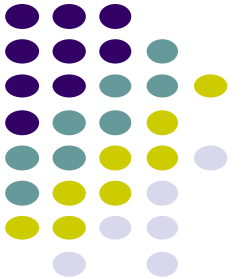


- 64 bit result
- Unsigned Long and Accumulate
- Signed Long and Accumulate
  - **UMULL{<cond>}{S} RdLo,RdHi,Rm,Rs**
  - **UMLAL{<cond>}{S} RdLo,RdHi,Rm,Rs**
  - **SMULL{<cond>}{S} RdLo, RdHi, Rm, Rs**
  - **SMLAL{<cond>}{S} RdLo, RdHi, Rm, Rs**

Using Link Register

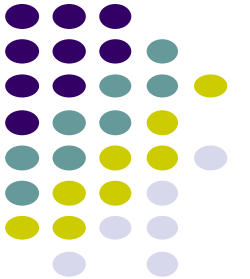
```
start:  MOV     r0, #10         ; Set up parameters
        MOV     rI, #3
        BL      add             ; Call subroutine
ret:     ...
        ...
        ...
         SWI     0xI I
 add: ADD       r0, r0, rI      ; Subroutine code
        BX       lr              ; Return from subroutine
```

Using Stack



```
subroutine    PUSH    {r5-r7,lr} ; Push working registers and lr
              ; code
                  BL      somewhere_else
              ; code
                  POP     {r5-r7,pc} ; Pop working registers and pc
```

What is not discussed?



1. Block Transfer
2. Writing C code, Compiling it for ARM
3. ARM and Thumb coexistence