Why do we need to know the instruction set

- **ARM and Thumb instruction sets were designed to give the best output from compilers**
  - Especially the Thumb Instruction set
- **Most design effort for many systems is focussed on compiled code and knowledge of the instruction set is not required**
- **But…..**
  - Embedded systems require initialisation code and interrupt routines
  - All systems require debugging – possibly at the instruction level
  - Performance gains can be made by writing assembler routines
  - Some features of the ARM architecture are not available with compilers

Agenda

- **Architecture v4T**

  Architecture v5TE

  Architecture v6

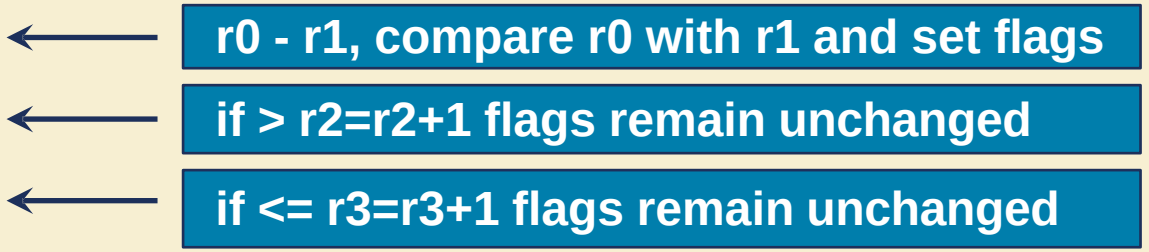
  Thumb

  Unified Assembler Language

Conditional Execution and Flags

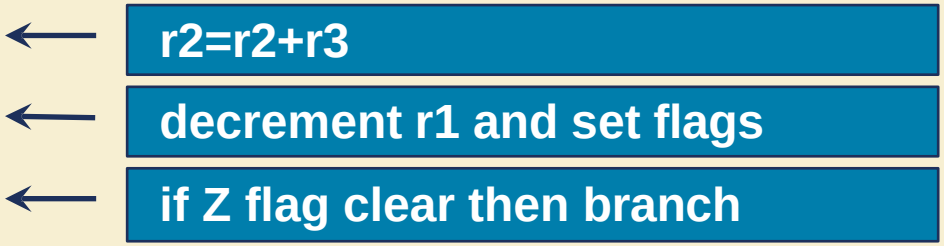
- **ARM instructions can be made to execute conditionally by post-fixing them with the appropriate condition code**
  - This can increase code density and increase performance by reducing the number of forward branches

```
CMP     r0, r1
ADDGT   r2, r2, #1
ADDLE   r3, r3, #1
```



- **By default, data processing instructions do not affect the condition flags but this can be achieved by post fixing the instruction (and any condition code) with an "S"**

```
loop
    ADD     r2, r2, r3
    SUBS    r1, r1, #0x01
    BNE     loop
```



Conditional execution examples

C source code

```
if (r0 == 0)
{
  r1 = r1 + 1;
}
else
{
  r2 = r2 + 1;
}
```

ARM instructions

unconditional

```
  CMP r0, #0
  BNE else
  ADD r1, r1, #1
  B end
else
  ADD r2, r2, #1
end
  ...
```

- 5 instructions
- 5 words
- 5 or 6 cycles

conditional

```
CMP r0, #0
ADDEQ r1, r1, #1
ADDNE r2, r2, #1
...
```

- 3 instructions
- 3 words
- 3 cycles

Condition Codes

- **The possible condition codes are listed below**
  - Note AL is the default and does not need to be specified

Suffix	Description	Flags tested
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Unsigned higher or same	C=1
CC/LO	Unsigned lower	C=0
MI	Minus	N=1
PL	Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 or Z=1
GE	Greater or equal	N=V
LT	Less than	N!=V
GT	Greater than	Z=0 & N=V
LE	Less than or equal	Z=1 or N=!V
AL	Always	

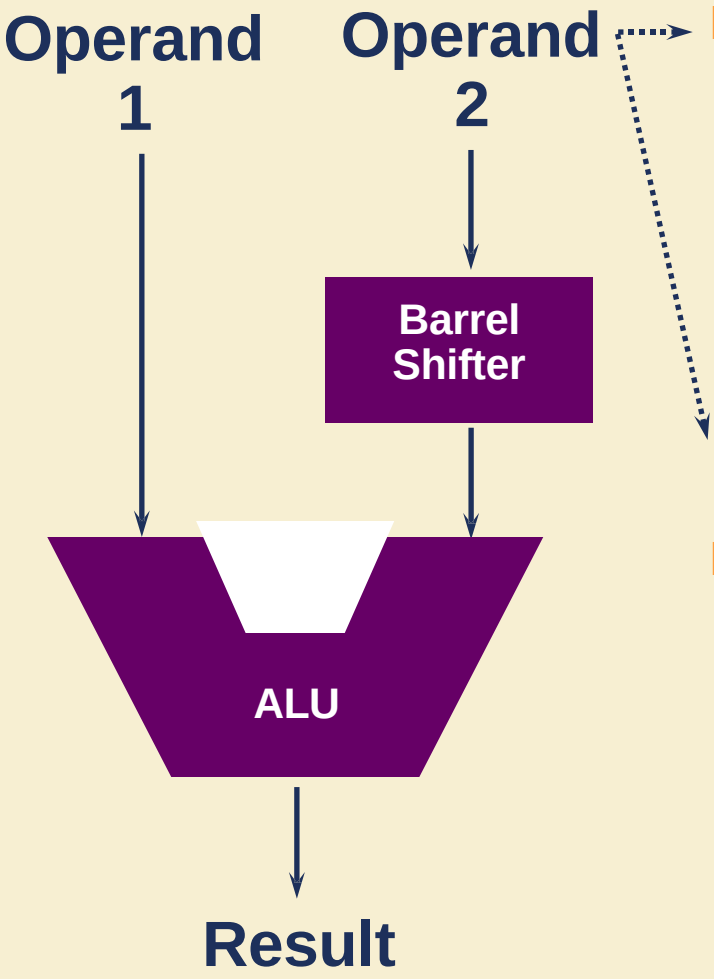
Data processing Instructions

- **Consist of :**
  - Arithmetic: `ADD` `ADC` `SUB` `SBC` `RSB` `RSC`
  - Logical: `AND` `ORR` `EOR` `BIC`
  - Comparisons: `CMP` `CMN` `TST` `TEQ`
  - Data movement: `MOV` `MVN`
- **These instructions only work on registers, NOT memory.**
- **Syntax:**

  `<Operation>{<cond>}{S} Rd, Rn, Operand2`

  - Operand 2 can be a register or an immediate value
    - `SUB r0, r1, r2`
    - `AND r1, r4, #0xFF`
  - Comparisons set flags only - they do not specify Rd
    - `CMP r0, r3`
  - Data movement does not specify Rn
    - `MOV r0, r1`
- **Operand 2 is sent to the ALU via 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
- Used for multiplication by constant
  - ADD r0, r5, r5 LSL 1 r0 = r5 x 3

Immediate value

- 8 bit number, with a range of 0-255.
  - Rotated right through even number of positions
- Allows increased range of 32-bit constants to be loaded directly into registers

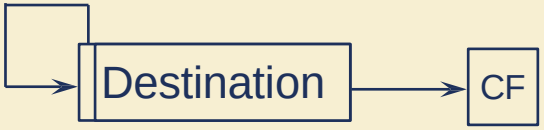
Shift Operations

- **These shift operations are used as part of data processing instructions.**
  - bits can be shifted from 0-31 places, typically without performance penalty

LSL: Logical Left Shift

Multiplication by a power of 2

ASR: Arithmetic Right Shift

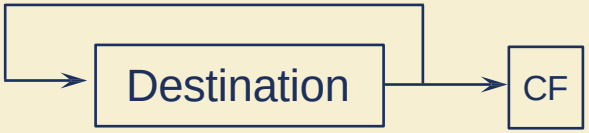


Division by a power of 2, preserving the sign bit

LSR: Logical Shift Right

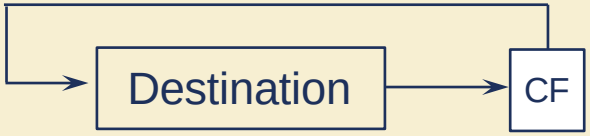
Division by a power of 2

ROR: Rotate Right



Bit rotate with wrap around from LSB to MSB

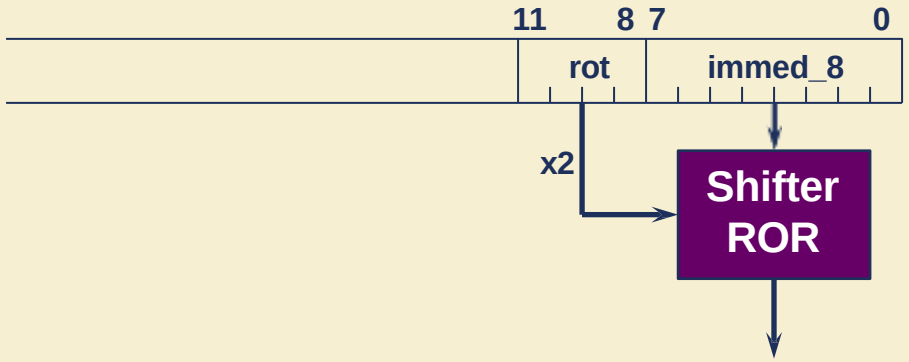
RRX: Rotate Right Extended



Single bit rotate with wrap around
from CF to MSB

Immediate constants

- **No ARM instruction can contain a 32 bit immediate constant**
  - All ARM instructions are fixed as 32 bits long
- **The data processing instruction format has 12 bits available for operand2**



Quick Quiz:

```
0xe3a004ff
MOV r0, #???
```

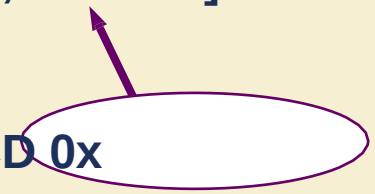
- **4 bit rotate value (0-15) is multiplied by two to give range 0-30 in steps of 2**
- **Rule to remember is**

  “8-bits rotated right by an even number of bit positions”

Loading 32 bit constants

- To allow larger constants to be loaded, the assembler offers a pseudo-instruction:
- LDR rd, =const
- This will either:
- Produce a MOV or MVN instruction to generate the value (if possible)
- or
- Generate a LDR instruction with a PC-relative address to read the constant from a literal pool (Constant data area embedded in the code)
- For example
- LDR r0, =0xFF => MOV r0, #0xFF
- LDR r0, =0x => LDR r0, [PC, #Imm12]

```
…
…
DCD 0x
```



- This is the mended way of loading constants into a register

Multiply and Divide

- **There are 2 classes of multiply - producing 32-bit and 64-bit results**
- **32-bit versions on an ARM7TDMI will execute in 2 - 5 cycles**

```
MUL r0, r1, r2              ; r0 = r1 * r2
MLA r0, r1, r2, r3          ; r0 = (r1 * r2) + r3
```

- **64 bit multiply instructions offer both signed and unsigned versions**
  - For these instruction there are 2 destination registers

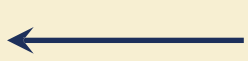
```
[U|S]MULL r4, r5, r2, r3    ; r5:r4 = r2 * r3
[U|S]MLAL r4, r5, r2, r3    ; r5:r4 = (r2 * r3) + r5:r4
```

- **Most ARM cores do not offer integer divide instructions**
  - Division operations will be performed by C library routines or inline shifts

Branch Instructions

- **Branch instructions have the following format:**
  - `B{L}{<cond>} label`
  - subroutine calls can be made by specifying the optional `{L}`
  - a 24 bit address offset field is part of the instruction encoding
    - On execution this is left shifted 2 places (since ARM instructions are always word aligned) to give a 26 bit value, thus giving a relative branch range of ± 32 MB
  - Causes a pipeline flush

```
    B    start
     .
     .
start
```



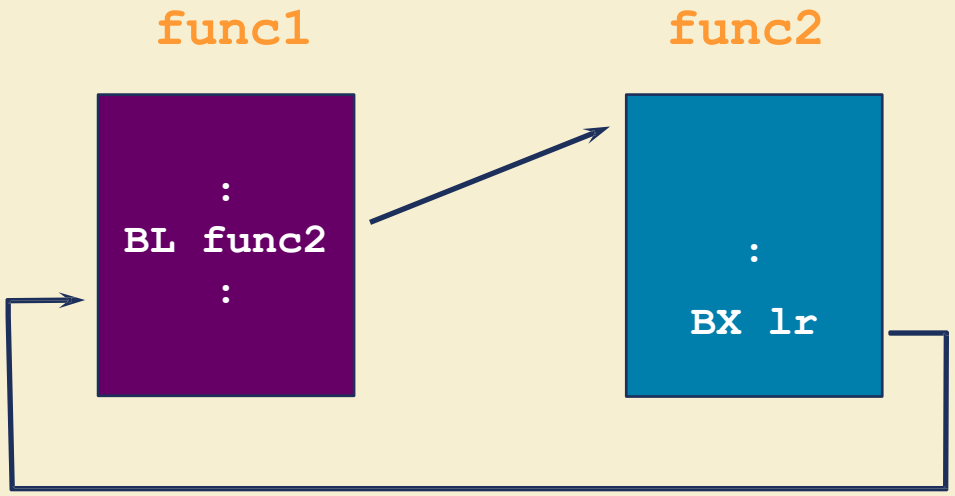
perform PC relative branch to label “start”

continue execution from here

Subroutines

- **Implementing a conventional subroutine call requires two steps:**
  - Store the return address
  - Branch to the address of the required subroutine
- **These steps are carried out in one instruction, `BL`**
  - The return address is stored in the link register (`lr/r14`)
  - Branch to an address anywhere within a +/- 32MB range
- **Returning is performed by restoring the program counter (`pc`) from `lr`**

```
void func1 (void)
{
        :
        func2();
        :
}
```



Quiz

1. Write an ARM instruction which will implement each of the following:

   a) `r0 = 16`

   b) `r0 = r1 / 16` `(signed numbers)`

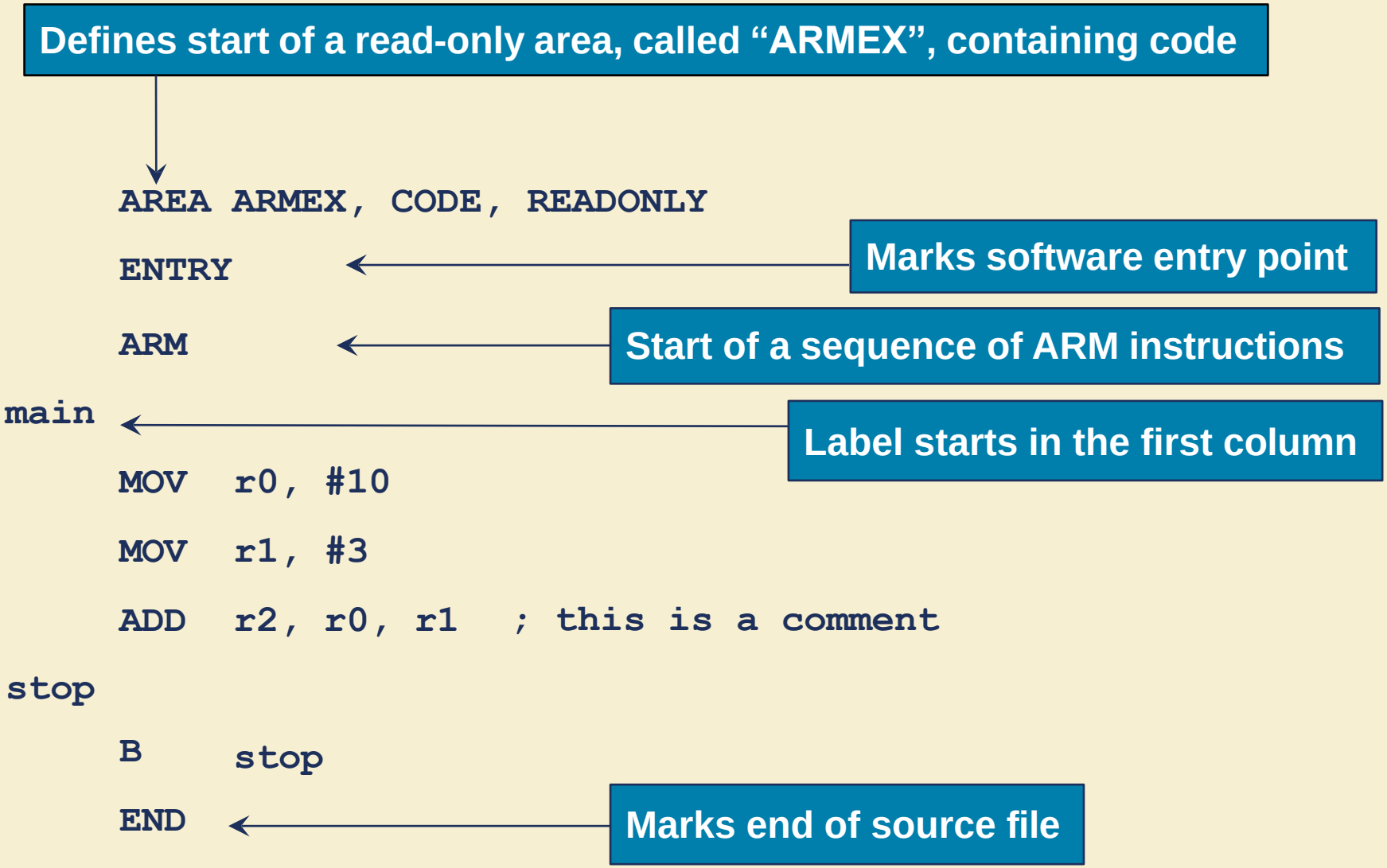
   c) `r1 = r2 * 3`

   d) `r0 = -r0`

2. What does the BIC instruction do?

3. Which data processing instructions always set the condition flags?

Example Source File



Workbook Session 1

- Basic assembler file and general data processing operations
- Loading constants into registers
- Conditional execution

Single register data transfer

`LDR`	`STR`	Word
`LDRB`	`STRB`	Byte
`LDRH`	`STRH`	Halfword
`LDRSB`		Signed byte load
`LDRSH`		Signed halfword load

- **Memory system must support all access sizes**

- **Syntax:**
  - `LDR{<cond>}{<size>} Rd, <address>`
  - `STR{<cond>}{<size>} Rd, <address>`

e.g. `LDREQB`

Address accessed

- **Address accessed by LDR/STR is specified by a base register with an offset**
- **For word and unsigned byte accesses, offset can be:**
  - An unsigned 12-bit immediate value (i.e. 0 - 4095 bytes)

    ```
    LDR r0, [r1, #8]
    ```

  - A register, optionally shifted by an immediate value

    ```
    LDR r0, [r1, r2]
    LDR r0, [r1, r2, LSL#2]
    ```

- **This can be either added or subtracted from the base register:**

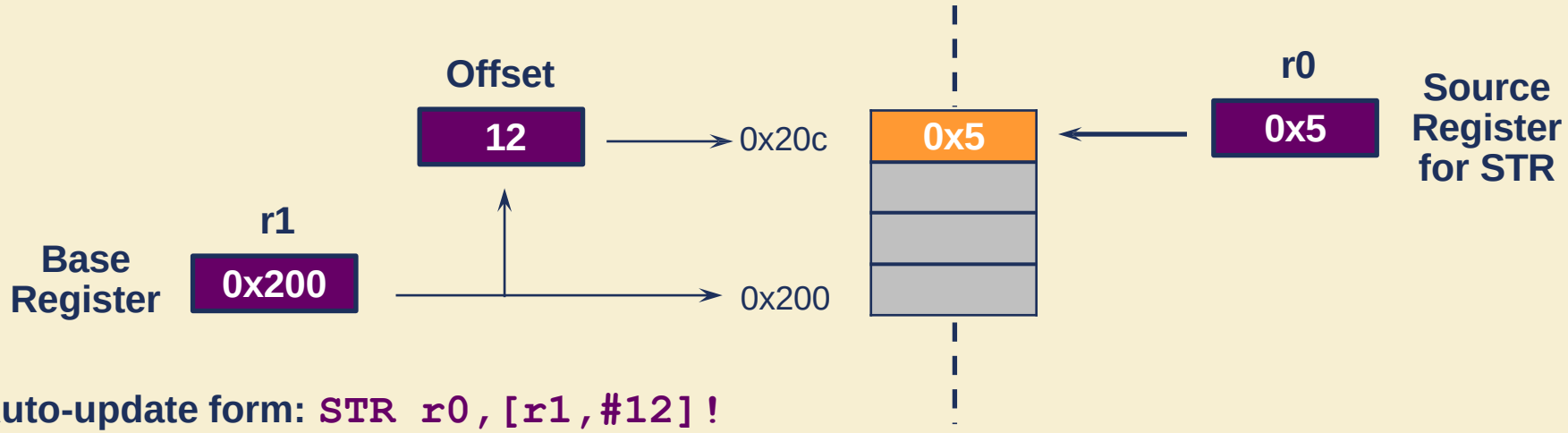
  ```
  LDR r0, [r1, #-8]
  LDR r0, [r1, -r2, LSL#2]
  ```

- **For halfword and signed halfword / byte, offset can be:**
  - An unsigned 8 bit immediate value (i.e. 0 - 255 bytes)
  - A register (unshifted)
- **Choice of *pre-indexed* or *post-indexed* addressing**
- **Choice of whether to update the base pointer (pre-indexed only)**

  ```
  LDR r0, [r1, #-8]!
  ```

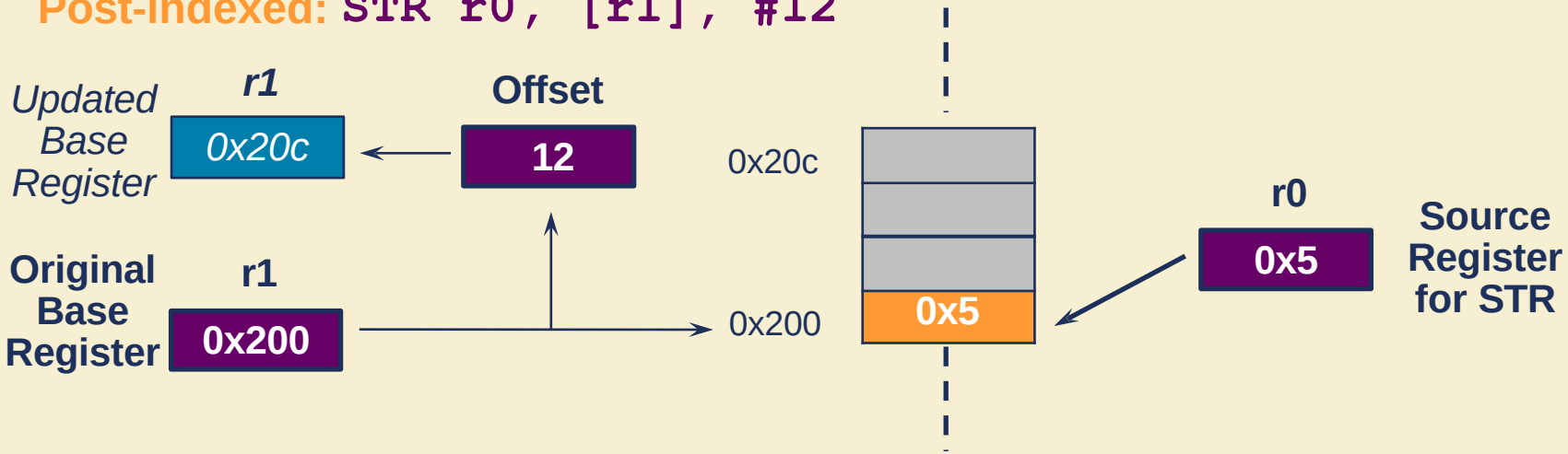
Pre or Post Indexed Addressing?

- Pre-indexed: `STR r0, [r1, #12]`



Auto-update form: `STR r0,[r1,#12]!`

- Post-indexed: `STR r0, [r1], #12`



Generating Branches with LDR

- **The ARM's branch instruction is limited to a range of ±32MB**
  - However branches can also be performed by loading address values directly into the **`PC (r15)`**
  - **`armasm`** provides pseudo instructions to make this easier

Assembler Code

```
LDR pc, =label  ; load address of label into PC
```

Branches anywhere within the 4GB address space are thus possible

Object Code

```
LDR pc, [pc, #n]
      .
      .
      .
DCD 0x
```

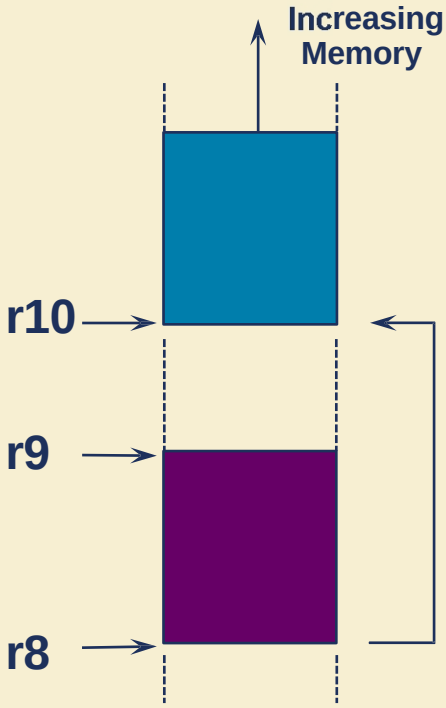
Literal pool address data

Memory Block Copying (1)

- **The use of base register updating enables simple copying routines to be written**
  - For example: The post-indexed variant could be used to copy a block of memory

```
; r8 points to start of source data
; r9 points to end of source data
; r10 points to start of destination data

loop    LDR     r0, [r8], #4   ; load 4 bytes
        STR     r0, [r10], #4  ; and store them
        CMP     r8, r9         ; check for the end
        BLT     loop           ; else loop
```

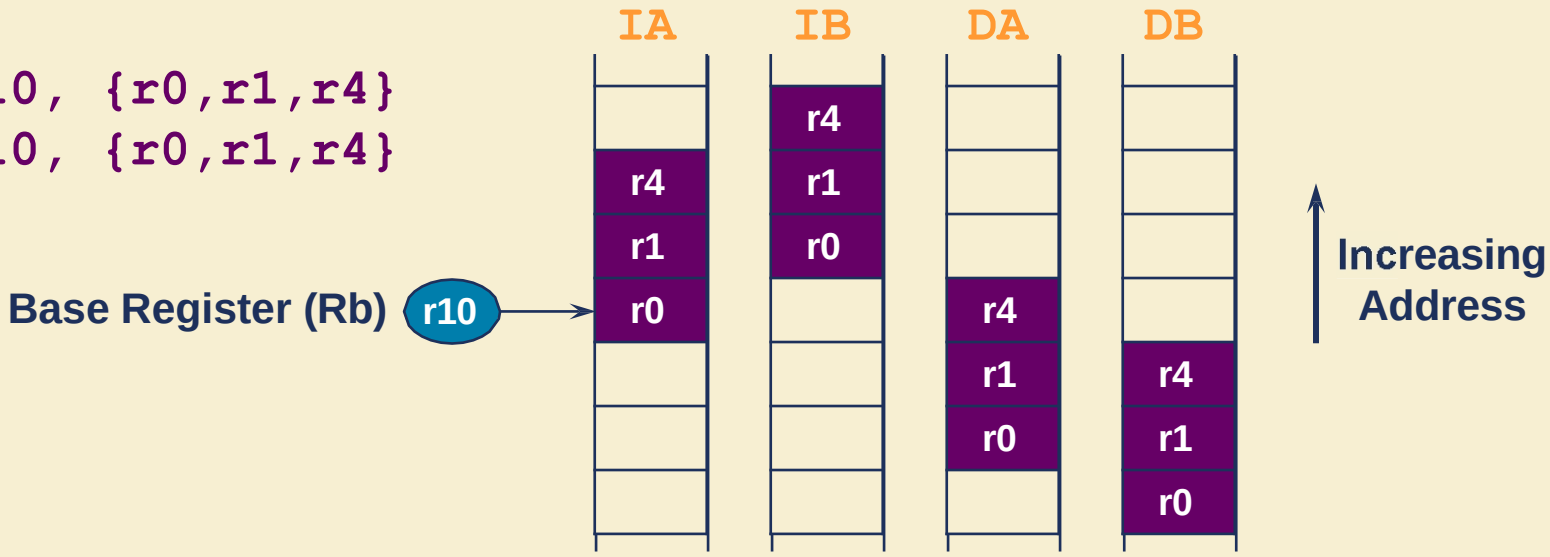


- **In this example 1 word is copied per iteration**

Load and Store Multiples

- **Syntax:**
  - **<LDM|STM>**{<cond>}<addressing_mode> Rb{!}, <register list>
- **4 addressing modes:**
  - **LDMIA / STMIA** increment after
  - **LDMIB / STMIB** increment before
  - **LDMDA / STMDA** decrement after
  - **LDMDB / STMDB** decrement before

```
LDMxx r10, {r0,r1,r4}
STMxx r10, {r0,r1,r4}
```



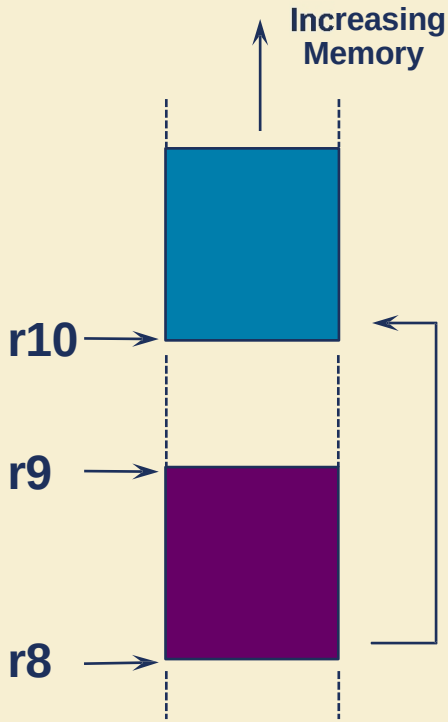
Memory Block Copying (2)

- As well as being used for stack operations, the `STM` / `LDM` instructions can perform block copying of memory
- For example

```
; r8 points to start of source data
; r9 points to end of source data
; r10 points to start of destination data

loop    LDMIA   r8!, {r0-r7}   ; load 32 bytes
        STMIA   r10!, {r0-r7}  ; and store them
        CMP     r8, r9         ; check for the end
        BLT     loop           ; and loop
```

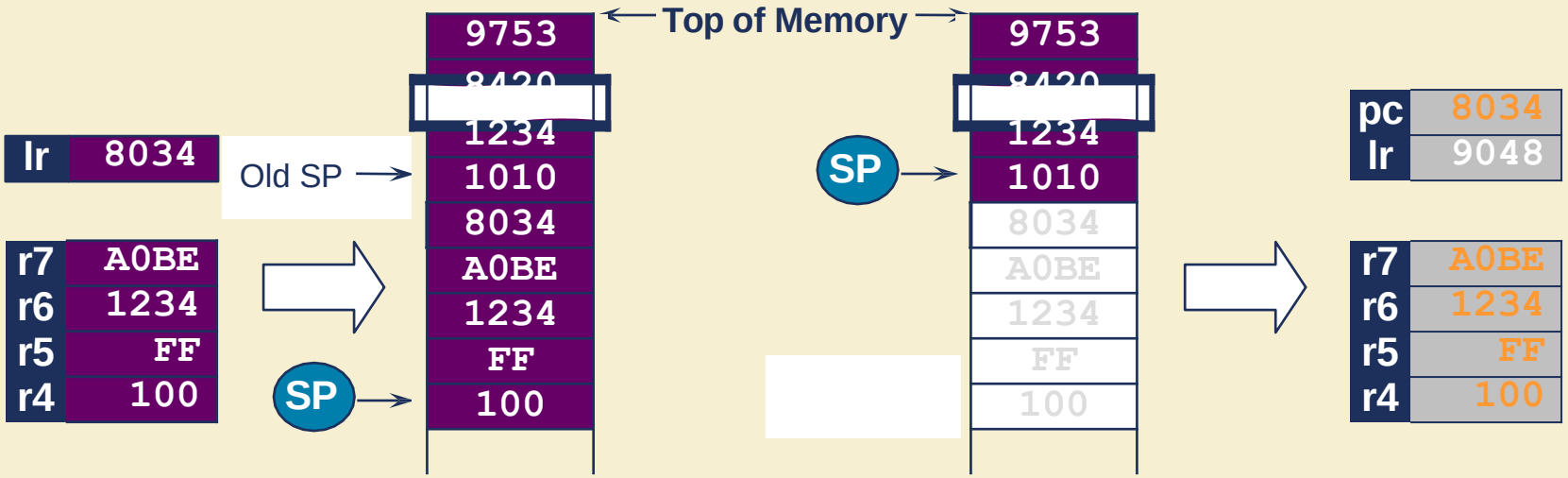
- In this example 8 words are copied per loop



Stacks

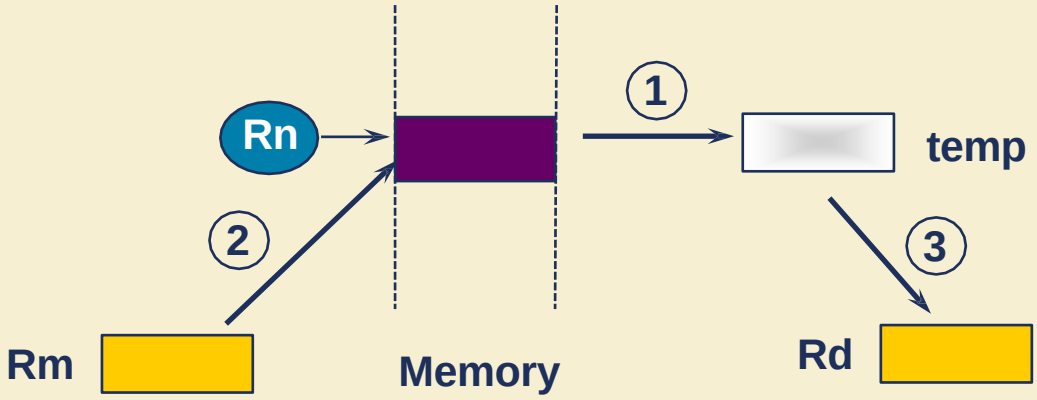
- **ARM stack operations are implemented block transfer instructions:**
  - **STMFD** (Push) Store Multiple - Full Descending stack [STMDB]
  - **LDMFD** (Pop) Load Multiple - Full Descending stack [LDMIA]
    - Note: Multiple registers will always be stacked in register order from lowest register to lowest memory location
    - The order registers are specified has no effect.

```
STMFD sp!, {r4-r7, lr}      LDMFD sp!, {r4-r7, pc}
```



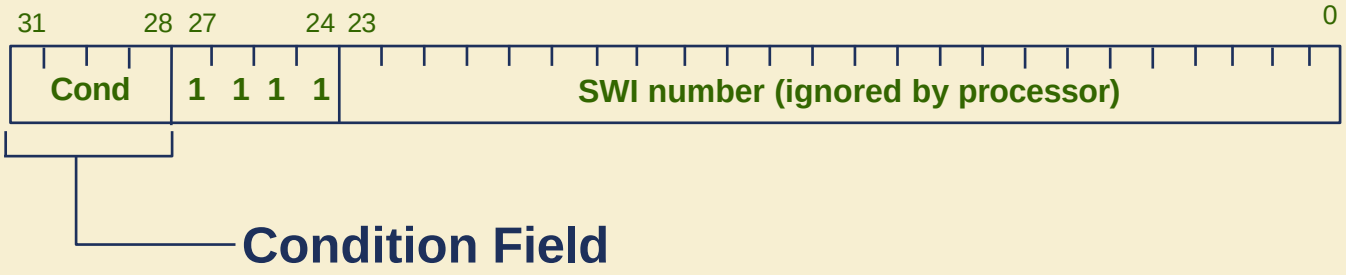
SWP

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



- Can be used to implement flags
- Cannot be generated from C using armcc - must use assembler

Software Interrupt (SWI)



- Causes an exception trap to the SWI hardware vector
- The SWI handler can examine the SWI number to decide what operation has been requested.
- By using the SWI mechanism, an operating system can implement a set of privileged operations which applications running in user mode can request.
- Syntax:
  - `SWI{<cond>} <SWI number>`

PSR access

- **MRS and MSR allow contents of CPSR / SPSR to be transferred to / from a general purpose register or take an immediate value**
  - MSR allows the whole status register, or just parts of it to be updated
- **Interrupts can be enable/disabled and modes changed, by writing to the CPSR**
  - Typically a read/modify/write strategy should be used:

```
MRS r0,CPSR        ; read CPSR into r0
BIC r0,r0,#0x80    ; clear bit 7 to enable IRQ
MSR CPSR_c,r0      ; write modified value to 'c' byte only
```

- **In User Mode, all bits can be read but only the condition flags (_f) can be modified**

Coprocessor Instructions

- **The ARM architecture supports 16 coprocessors**
- **The instructions for each coprocessor occupy a fixed part of the ARM instruction set**
  - If the appropriate coprocessor is not present in the system, an undefined instruction exception occurs
- **There are three types of coprocessor instruction**
  - Coprocessor data processing
    - **`CDP`** : Initiate a coprocessor data processing operation
  - Coprocessor register transfers
    - **`MRC`** : Move to ARM register from coprocessor register
    - **`MCR`** : Move to Coprocessor register from ARM register
  - Coprocessor memory transfers
    - **`LDC`** : Load coprocessor register from memory
    - **`STC`** : Store from coprocessor register to memory

Quiz 2

1. What assembler directive should start every assembler source file?

2. What instructions can be used to return from a leaf subroutine call?

3. What instructions should be used to enable or disable IRQ interrupts?

4. What instructions can be used to e the ± 32MB limitation of the standard ARM Branch instruction?

5. For what might you use the SWP instruction?

Workbook session 2

- Subroutines and stacks
- Block copying

Agenda

- Architecture v4T
- **Architecture v5TE**
- Architecture v6
- Thumb
- Unified Assembler Language

ARM Architecture v5TE

- **Architecture v5TE contains full v4T ARM and Thumb instruction sets, plus:**
  - Improved support for ARM / Thumb interworking
    - Covered in ARM / Thumb Interworking module
  - Count Leading Zeros instruction
  - Packed half-word signed multiplication instructions
  - Support for saturated mathematics
  - Addition of Q flag to PSRs
  - Double-word Load / Store instructions
  - Breakpoint instructions (ARM and Thumb)
  - Cache Preload instruction

Count Leading Zeros

- **`CLZ{cond} Rd, Rm`**
  - returns number of binary zero bits before the first binary one bit in a register value
  - source register is scanned from most significant bit to least significant bit
  - executes in 1-cycle (ARM9E-S/ARM102x)
  - result is 32 if no bits set, zero if bit 31 is set
  - Used in software divide and floating point routines.
- **Left shift of Rm by Rd will normalize Rm**
- **Signed normalize requires 1 extra cycle**

```
EOR R1, R0, R0, LSL #1
CLZ R1, R1
MOV R0, R0, LSL R1
```

R0 = 0000 0010 1110 1101...0

```
CLZ R1, R0
```

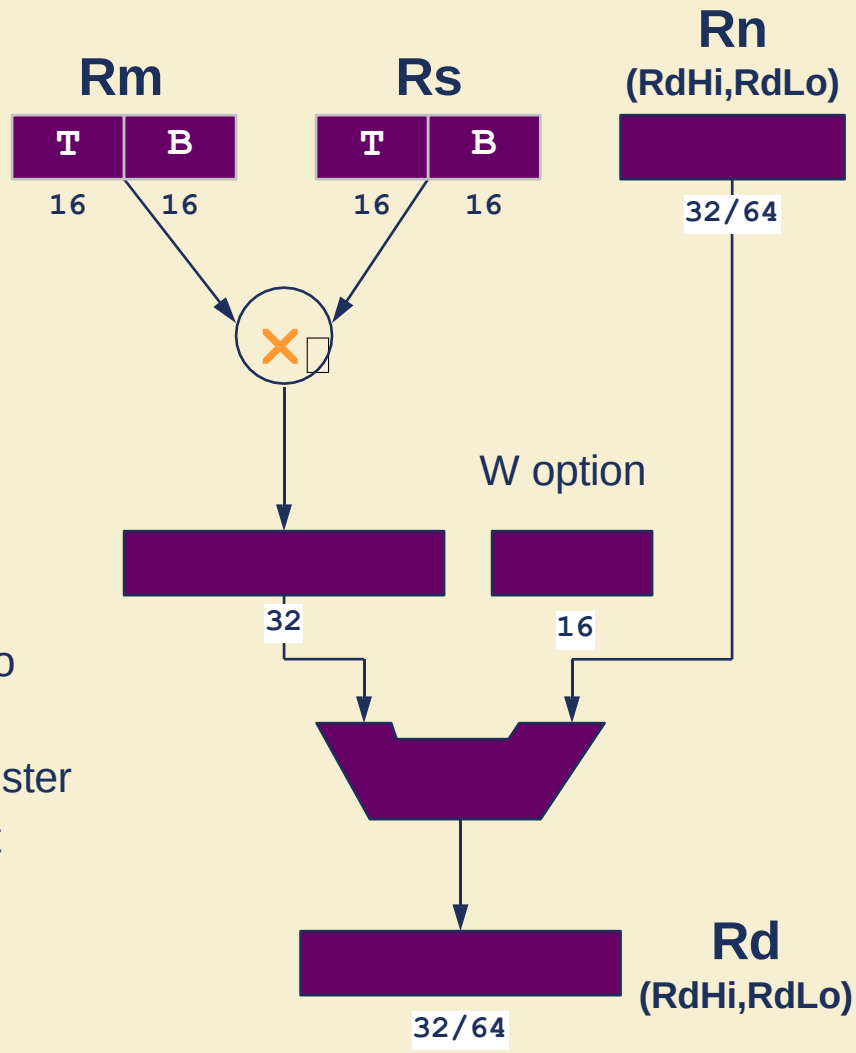
R1 = 0x6

```
MOV R0, R0 LSL R1
```

Rm = 1011 1011 0100 0000...0

Signed Multiply Operations

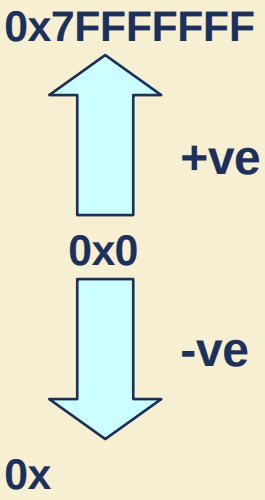
- **SMULxy{cond} Rd, Rm, Rs**
- **SMULWy{cond} Rd, Rm, Rs**
- **SMLAxy{cond} Rd, Rm, Rs, Rn**
- **SMLAWy{cond} Rd, Rm, Rs, Rn**
- **SMLALxy{cond} RdLo, RdHi, Rm, Rs**



- Q flag is affected for `SMLA` instructions (but no saturation)
- `x, y` selects either `T`op or `B`ottom half of register
- `W` selects the upper 32 bits of a 48-bit product
- Do not affect NZCV (‘S’ is not allowed)

Saturated Maths Instructions

- Saturated maths
- Adding 1 to 0x7FFFFFFF causes a transition from a positive value to a negative value
- Subtracting 1 from 0x causes a transition from a negative value to a positive value



- Saturation is required by several DSP algorithms
- G.723.1 - VoIP
- AMR - Adaptive MultiRate

- QSUB{cond} Rd, Rm, Rn ;Rd = saturate(Rm - Rn)
- QADD{cond} Rd, Rm, Rn ;Rd = saturate(Rm + Rn)
- QDSUB{cond} Rd, Rm, Rn ;Rd = saturate(Rm - saturate(Rn 2))
- QDADD{cond} Rd, Rm, Rn ;Rd = saturate(Rm + saturate(Rn 2))

- Q flag will be set if saturation occurs during these instructions

Load / Store Double Registers

`LDR/STR{<cond>}D <Rd>, <addressing_mode>`

- Transfer two adjacent words in memory to / from any of the registers pairs (r0,r1), (r2,r3), (r4,r5), (r6,r7), (r8,r9), (r10,r11) or (r12,r13)
- Rd specifies the even numbered register - the immediately following odd numbered register is used for the second transfer
- Use same addressing modes as LDRH/STRH
- Address is that of the lower of the two words loaded by the `LDRD` instruction - the address of the higher word is generated by adding 4 to this address
- Address must be doubleword (8-byte) aligned

Breakpoint & Cache Preload

- **Breakpoint Instruction -** `BKPT <#imm16>`
  - Set up by debug agent in RAM in system
  - Immediate value is ignored by the processor
  - Execution of this instruction will either cause a prefetch abort or cause the processor to enter debug state (depends on the core design & configuration)

- **Cache Preload Instruction-** `PLD [Rn, <offset>]`
  - Offset can be
    - An unsigned 12-bit immediate value (ie 0 - 4095 bytes).
    - A register, optionally shifted by an immediate value
  - Tells the memory system that an access to the data at a specified address is likely to occur soon
  - PLD is a hint instruction
    - On implementations that do not support this operation it will behave as a NOP

Quiz 3

1. What does the CLZ instruction do?

2. What is the result of the following instruction?

   r1 = 0x7FFFFF00 r2 = 0x

   QADD r0, r1, r2

3. Is the following instruction valid?

   LDRD r7, [r2, 0x100]

4. What is the affect of the following instruction?

   SMULBT r0, r1, r2

Workbook Session 3

- Using saturated maths / packed halfword multiplication instructions
- Accessing the CPSR

Agenda

- Architecture v4T
- Architecture v5TE
- **Architecture v6**
- Thumb
- Unified Assembler Language