Embedded Systems 1 (ENCE 361)

Lecture 33

ARM Assembly Language Programming

By: Dr. Steve Weddell

- Why use Assembler?
- Machine/program representation
- Addressing Modes
- Multiply and Shift Instructions
- Conditional execution instructions
- Reference and Exercises.

Why use Assembler?

1. To optimise your program in code critical sections.
2. Test the efficiency of your compiler.

Why use a high-level languages, e.g., C?

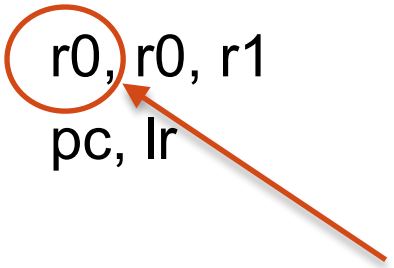
1. To reduce develop time.
2. Mnemonics are more cryptic than HLL statements and constructs.
3. C encourages programmers to modularise.

My preference? C, with ASM subroutines or modules to boost performance for real-time signal processing apps.

Basically, when using assembler you're in control!

ARM assembly language programming

LABEL	OPCODE (mnemonic)	OPERANDS	COMMENTS
Start:			
	MOV	r0, #15	; immediate operand
	MOV	r1, #$43	;
	BL	Myfunc	; call to subroutine
			; and save retrn addr
Here:	B	Here	; endless loop
Myfunc:			
	ADD	r0, r0, r1	; add two operands
	MOV	pc, lr	; **return from subr**.



Types of ARM assembly instructions

- There are three types of ARM assembly instructions:
  1. Data processing
  2. Data movement
  3. Control flow.
- Requirements for data processing instructions:
  - Operands are 32-bit, i.e., from registers
  - Register result is 32-bit.
  - Supported by a 3-address architecture - 2 source, 1 destination operand, e.g., `r0 = r1 + r2`.
- Both signed and unsigned (2's comp) data supported.
- Status bits may be **optionally** updated by appending an 'S' to the mnemonic.

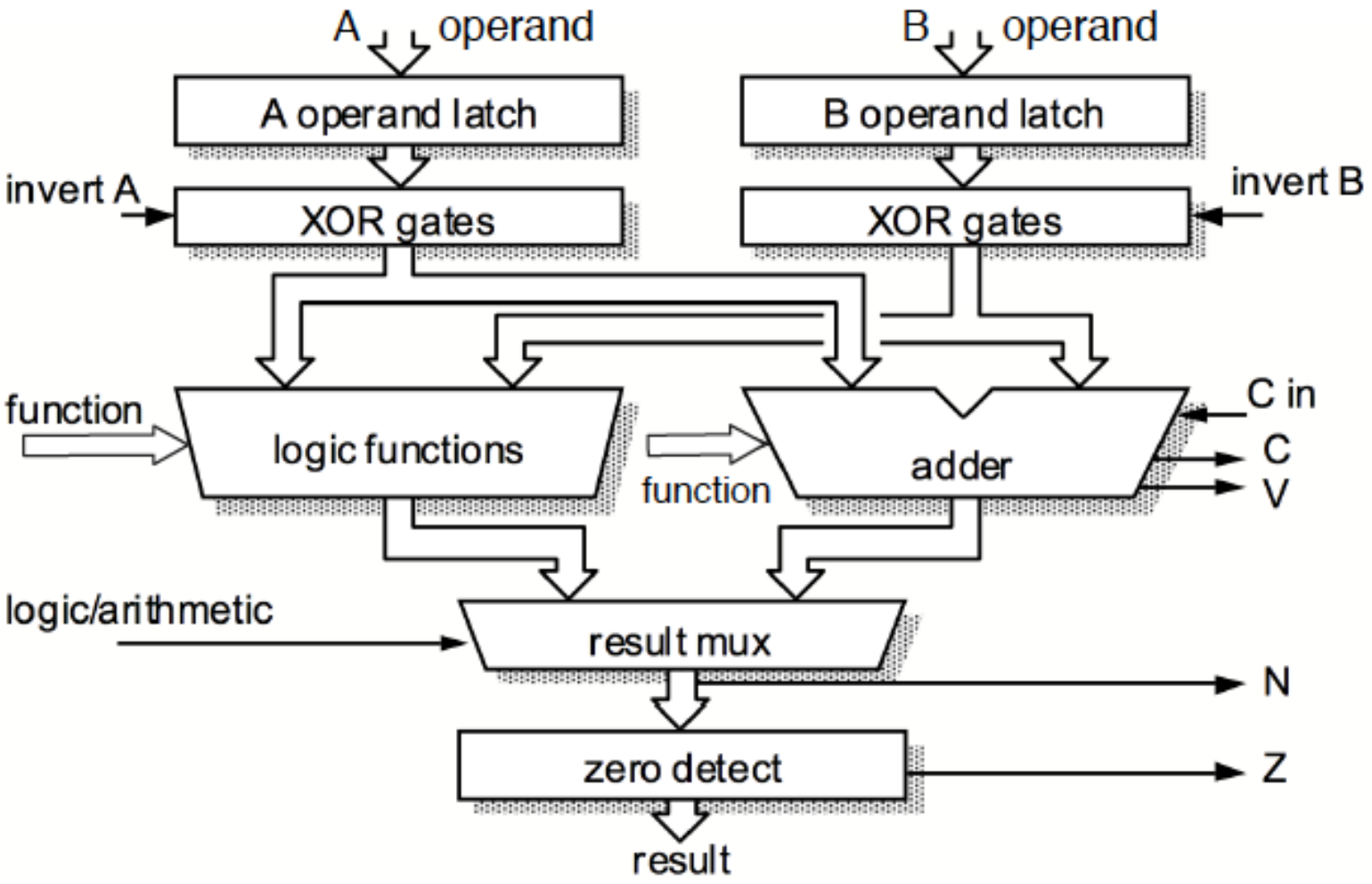
A simple example is the ARM6 ISA…

Data Processing Instructions

Opcode [24:21]	Mnemonic	Meaning	Effect
0000	AND	Logical bit-wise AND	Rd := Rn AND Op2
0001	EOR	Logical bit-wise exclusive OR	Rd := Rn EOR Op2
0010	SUB	Subtract	Rd := Rn - Op2
0011	RSB	Reverse subtract	Rd := Op2 - Rn
0100	ADD	Add	Rd := Rn + Op2
0101	ADC	Add with carry	Rd := Rn + Op2 + C
0110	SBC	Subtract with carry	Rd := Rn - Op2 + C - 1
0111	RSC	Reverse subtract with carry	Rd := Op2 - Rn + C - 1
1000	TST	Test	Scc on Rn AND Op2
1001	TEQ	Test equivalence	Scc on Rn EOR Op2
1010	CMP	Compare	Scc on Rn - Op2
1011	CMN	Compare negated	Scc on Rn + Op2
1100	ORR	Logical bit-wise OR	Rd := Rn OR Op2
1101	MOV	Move	Rd := Op2
1110	BIC	Bit clear	Rd := Rn AND NOT Op2
1111	MVN	Move negated	Rd := NOT Op2

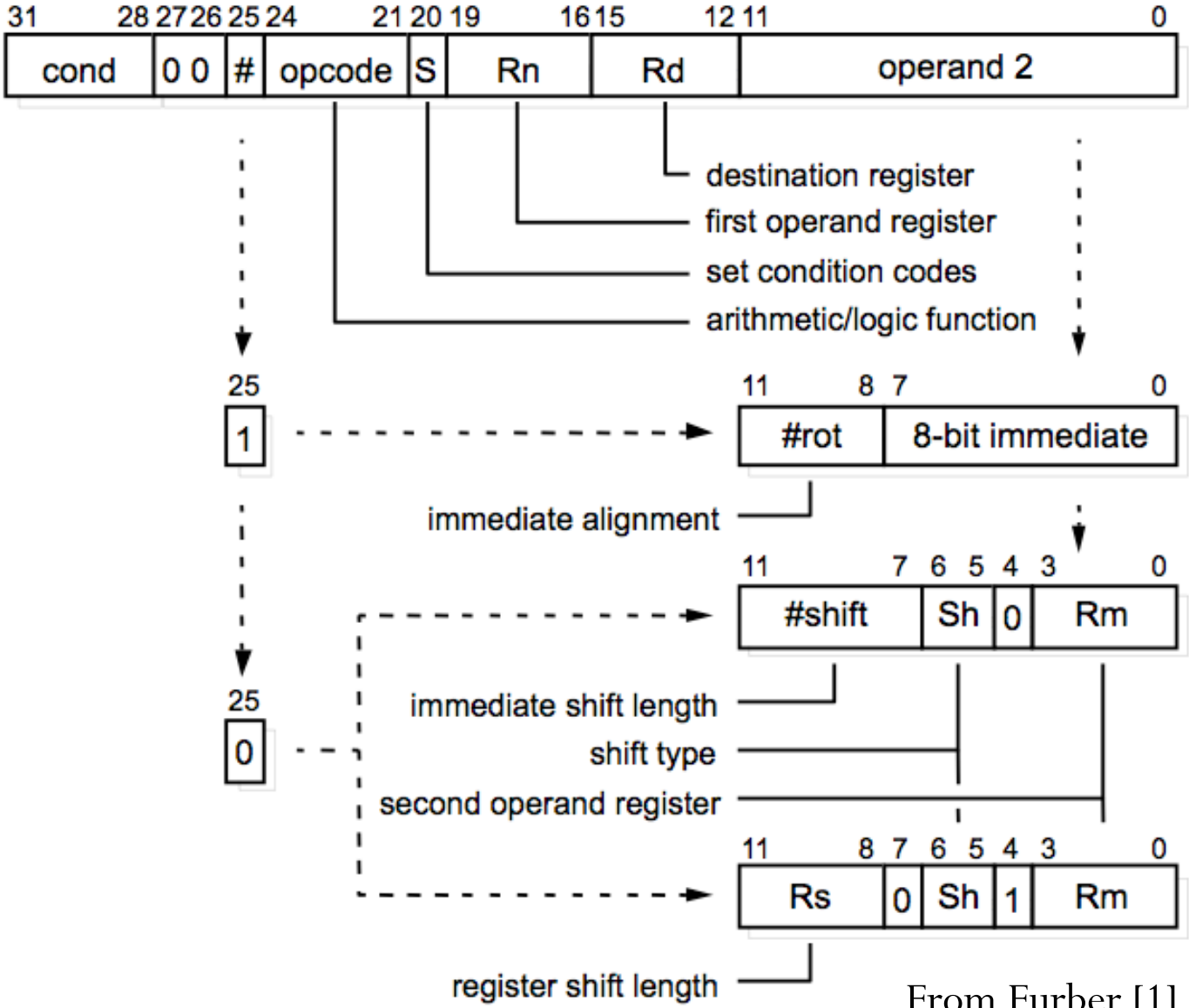
NB: i) 10 of the above instructions use the ALU; ii) 2 move data, and iii) 4 test/compare data that set/clear condition code registers.

Executing ARM Data Processing Instructions



ARM6 ALU (from Furber, 2000)

ARM Data Processing Instruction Format



From Furber [1]

Data transfer instructions - single register

- Data transfer instructions can be either:
  - Single register load/store instructions
  - Multiple register load/store instructions
  - Single register swap instructions
- A value is used in a base register and this forms a memory address for data load or data storage requirements.
- Examples:

```
LDR     r0, [r1]        ; r0 = mem[r1]
STR     r0, [r1]        ; mem[r1] = r0
```

- Also refer to register indirect addressing.

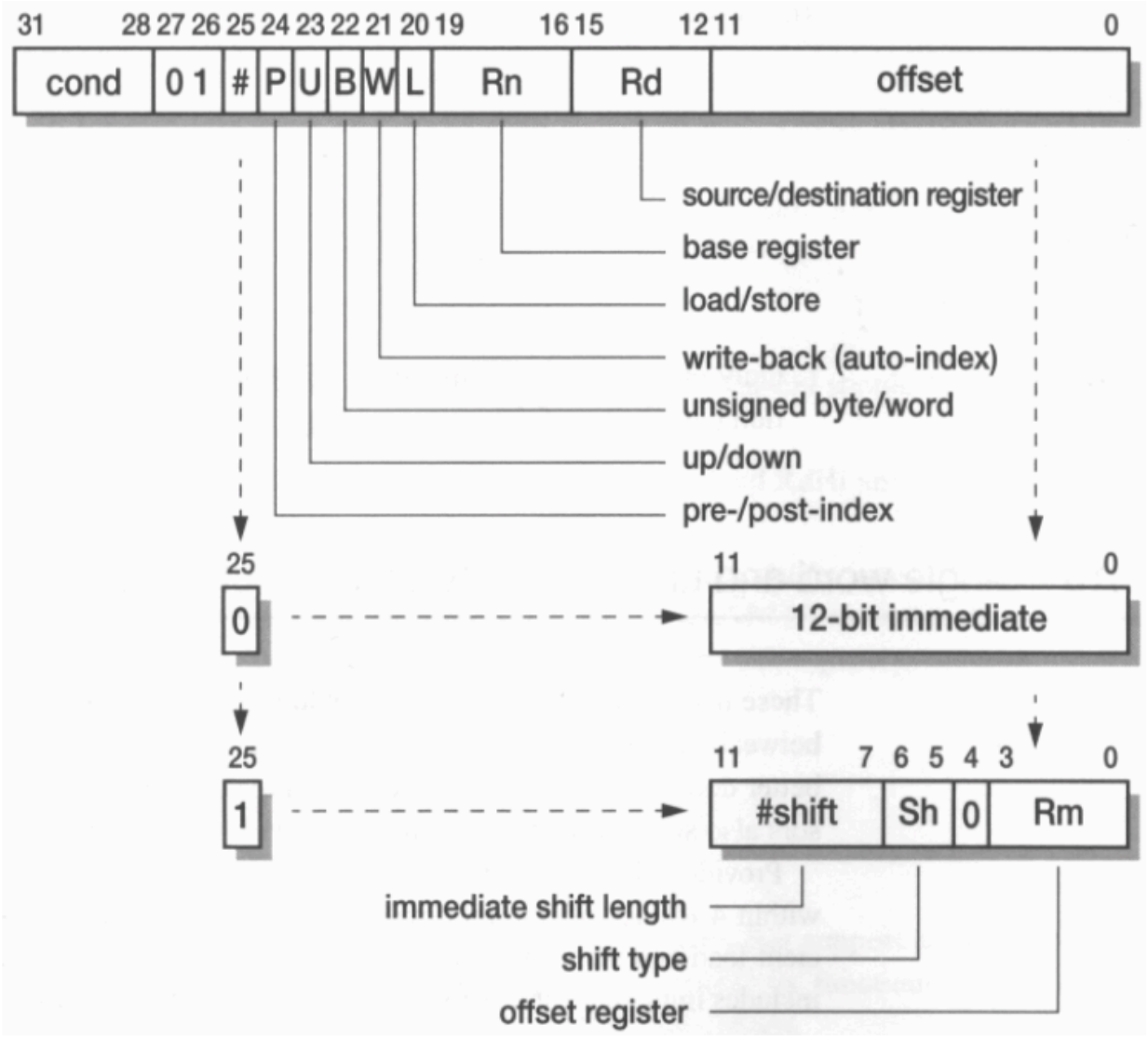
Control flow instructions

- Determines which instruction executes next.
- Most common is the *branch* (B) instruction.
  - Example: `Loop  B  Loop`
- Conditional branches (BNE, BCC, BLO, etc.), test the condition codes, e.g., the Z bit, to determine if a branch should be taken.

```
        MOV   r0, #0      ; for (i=0; i<10; i++)
LOOP    …                 ; {…};
        ADD   r0, r0, #1
        CMP   r0, #10
        BNE   LOOP
```

Word & unsigned byte Data Transfer Instructions

- Single words or unsigned bytes can be transferred between memory and registers using these instructions.
- Example instructions include load register, "LDR" and store register "STR".
- An effective address typically includes a base register Rn plu offset.



From Furber [1]

Branch Conditions

Branch	Interpretation	Normal uses
B	Unconditional	Always take this branch
BAL	Always	Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set	Arithmetic operation gave carry-out
BHS	Higher or same	Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

Data transfer instructions & specialised addressing modes

- Register indirect:
  - Used for transfer instructions
  - Base plus offset, and base plus index addressing
  - A register is used as a base address as either a source and/or destination operand.
  - Examples: LDR r0, [r1] STR r0, [r2]
- Base plus Offset:
  - A base register provides an approximate memory location for the source/ destination operand. The addition of an index (optionally automatic increment, '!') is provided, both pre- and post-increment.
  - Examples:

```
LDR  r0, [r1, #4]        ; r0 = mem[r1 + 4] (pre)
LDR  r0, [r1], #4        ; r0 = mem[r1]; r1 = r1 + 4 (post)
```

More on control flow instructions

- Conditional branch instructions work with condition test instructions, e.g., CMP r0, r1, by setting or clearing the respective condition flags (also called status flags) in the CPSR.
- A conditional branch instruction immediately following a test condition instruction, e.g., CMP, will change the program flow if the test condition is met.
- 16 conditional branch conditions are available.
- The status flags in the CPSR can be optionally set AFTER instruction execution, by appending an 'S' on an instruction mnemonic, e.g., immediately preceding a conditional branch.
- For example:

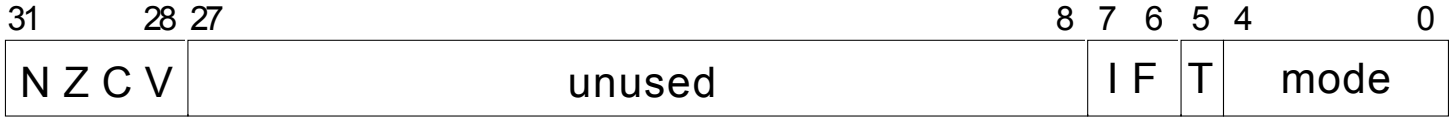
```
SUBS  r0, r0, #1          or      SUB  r0, r0, #1
BEQ   ..                          CMP  r0, #0
                                  BEQ  …
```

More on Comparison Operations

- The following is an extract from the list of Data Processing Instructions on Slide-5:

```
CMP     r1, r2     ; set cond. codes (cc) on r1 - r2
CMN     r1, r2     ; set cc. on r1 + r2
TST     r1, r2     ; set cc. on r1 and r2
TEQ     r1, r2     ; set cc. on r1 xor r2
```

- Only the cc. bits in the CPSR are set or cleared by executing these instructions.



N = 1, if MSB of (r1 - r2) is '1'; Z = 1 if (r1 - r2) = 0;

C = 1, if r1 & r2 are unsigned and r1 < r2; V = 1 if (r1, r2) are unsigned and r1 < r2

Multiply & Shift Instructions

- Multiply, and multiply and accumulate (commonly know as a MAC), instructions are supported (MUL, MLA resp.).
- These instructions differ from other instructions:
  - Immediate 2nd operands not supported.
  - The destination register cannot be the same as 1st operand
  - If the 'S' bit is set, the 'V' (oVerflow) flag is preserved but the 'C' (Carry) flag is meaningless.
- Both short and long multiply instructions are supported.

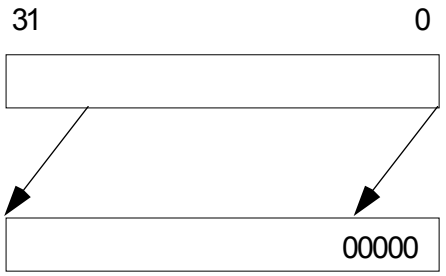
  Typical multiply examples:

```
MUL     r4, r3, r2          ; r4 = r3 x r2
MLA     r4, r3, r2, r1      ; r4 = (r3 x r2 + r1) note 4 operands!
```

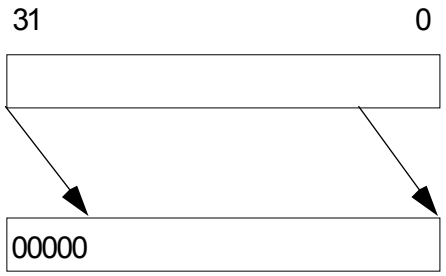
- Multiplies can be implemented with shifts, e.g., r0 x 35:

```
ADD     r0, r0, r0, LSL #2          ; r0' = 5 x r0
RSB     r0, r0, r0, LSL #3          ; r0" = 7 x r0'
```

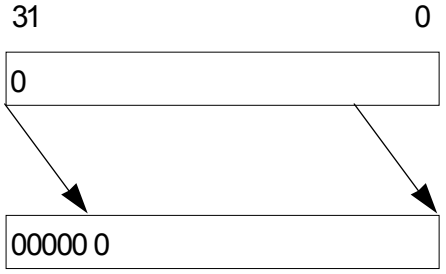
Shift Operations



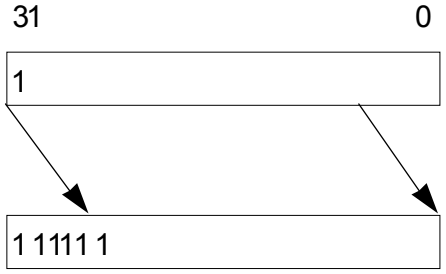
LSL #5



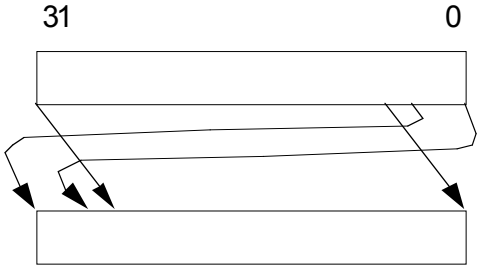
LSR #5



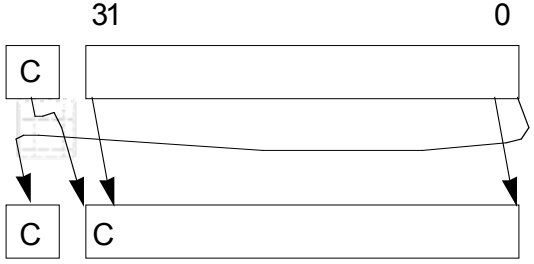
ASR #5 , positive operand



ASR #5 , negative operand



ROR #5



RRX

Conditional Execution

- Conditional instruction execution is an advanced feature supported by ARM cores.
- Any instruction (or group of instructions) may be conditionally executed by appending a condition of execution code to the suffix of the instruction mnemonic.
- Examples:

```
START:
  CMP r0, #5
  BEQ NXT       ;  if (r0 != 5) {
  ADD r1, r1, r0 ;    r1' = r1 + r0
  SUB r1, r1, r2 ;    r1 = r1' - r2
NXT:            ;  }
```

```
START:
  CMP    r0, #5     ; if (r0 != 5) {
  ADDNE  r1, r1, r0 ;   r1' = r1 + r0
  SUBNE  r1, r1, r2 ;   r1 = r1' - r2
  …                 ; }
```

Conditional instruction execution

- This example shows how to use conditional instruction execution to cascade multiple, conditional statements:

```
;; If (x==y) && (m==n) j++
        CMP     r0, r1      ; r0=x, r1=y
        CMPEQ   r2, r3      ; r2=m, r3= n
        ADDEQ   r4, r4, #1  ; j = j + 1
```

- Conditional execution is only efficient if the conditional sequence is three or less instructions. Use a branch conditional if your conditional sequence is 3 or more.
- Given the above example, it is also possible to have an else condition (to the two condition expressions shown). In that case, the xxxNE can be used, where xxx is an instruction mnemonic.

More on conditional Execution

Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

In-line assembly programming

- Why? It is a fast way of executing one or more assembly instructions in C.
- The compiler passes each statement directly, and also translates the parameters directly to the assembler.
- Depending on the compiler you use, you may need to precede each statement with the *volatile* operator.
- Be careful when using this. C has its own way of doing things and writing directly to registers such as r13, 14, or r15 can have dire consequences.

```
asm(" .arm
     mov r0,r1");     // NOTE: This
// worked on earlier GCC compilers

asm(" .arm\n"
    " mov r0,r1\n");   // NOTE: This
// works on all versions of GCC.
```

OR

```
asm(" .arm\n\
     mov r0,r1\n");   // NOTE: This also
// works on all versions of GCC.
```

Note: you must include a space between the first quotation mark and your instruction / pseudo-op.

References

[1] Furber, S., *ARM system-on-chip architecture*, 2nd Ed., Addison-Wesley, 2000.

[2] Atmel Corporation, *AT91 ARM Thumb-based Microcontrollers Datasheet*, Preliminary, November, 2006.

[3] Cockerell. P., *ARM Assembly Language Programming*, Computer Concepts Ltd., http://www.peter-cockerell.net/aalp/html/frames.html

Homework

1. Read §3.1-§3.5 (pp. 50-73) & §6.11(pp.186-187) from Furber [1] (the Engineering and Physical Sciences libraries have multiple copies on a short term loan).
2. Using conditional execution instructions, write the assembly code for:

   if ((x == y) && (m == n) j++;
3. What is the difference between *ASR* and *LSR* instructions, and *ASL* and *LSL* instructions?
4. Study the source code example on Learn.
5. What does ".arm" mean, as it is shown within each "asm(…)" code sequence on slide 20?