RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

ECE-332:437
DIGITAL SYSTEMS DESIGN (DSD)

Fall 2016 – Lecture 10
Architecture (ARM)

Nagi Naganathan
November 10, 2016

Announcements/Reminders -November 10, 2016

- No Quiz today
- Final Quiz on November 17, 2016
- Memory and I/O Sub-Systems – November 17, 2016
- Mid Term 2 – Dec 1 (Chapters 6, 7, 8)
- Finals – To Be Announced (Cumulative)

Topics to cover today – November 10, 2016

- Lecture 9 - Microarchitecture – Recap
- Lecture 10 – ARM Architecture – From additional text book

ARM Architecture

- Developed in the 1980's by Advanced RISC Machines – now called ARM Holdings
- Nearly 10 billion ARM processors sold/year
- Almost all cell phones and tablets have multiple ARM processors
- Over 75% of humans use products with an ARM processor
- Used in servers, cameras, robots, cars, pinball machines,, etc.

ARM Ltd

- Founded in November 1990
  - Spun out of Acorn Computers
  - Initial funding from Apple, Acorn and VLSI
- Designs the ARM range of RISC processor cores
  - Licenses ARM core designs to semiconductor partners who fabricate and sell to their customers
  - ARM does not fabricate silicon itself
- Also develop technologies to assist with the design-in of the ARM architecture
  - Software tools, boards, debug hardware
  - Application software
  - Bus architectures
  - Peripherals, etc

Intellectual Property

- **ARM provides hard and soft views to licencees**
  - RTL and synthesis flows
  - GDSII layout
- **Licencees have the right to use hard or soft views of the IP**
  - soft views include gate level netlists
  - hard views are DSMs

Huge Range of Applications

Development of the ARM Architecture

1

2

3

Early ARM architectures

4

Halfword and signed halfword / byte support

System mode

SA-110

SA-1110

4T

Thumb instruction set

ARM7TDMI

ARM9TDMI

ARM720T

ARM940T

5TE

Improved ARM/Thumb Interworking

CLZ

Saturated maths

DSP multiply-accumulate instructions

ARM1020E

XScale

ARM9E-S

ARM966E-S

5TEJ

Jazelle

Java bytecode execution

ARM9EJ-S

ARM926EJ-S

ARM7EJ-S

ARM1026EJ-S

6

SIMD Instructions

Multi-processing

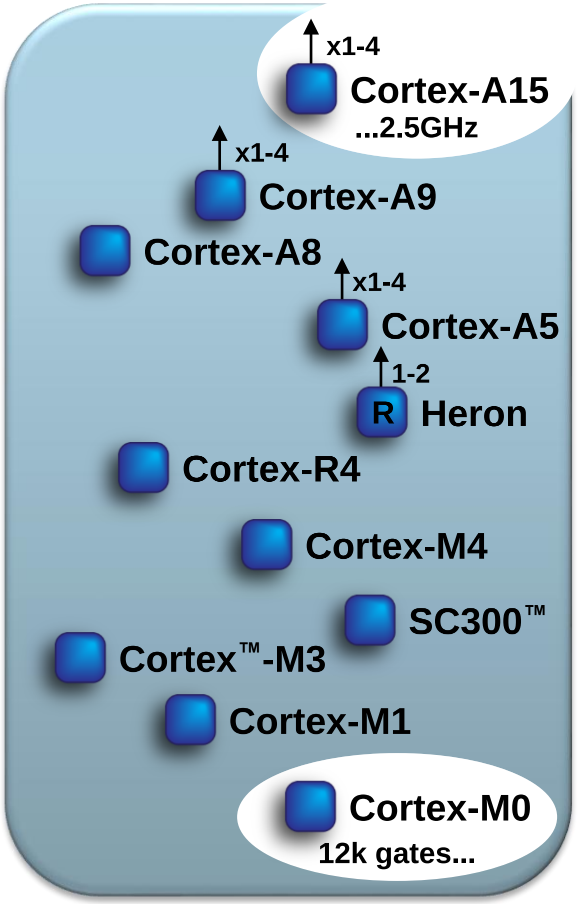
V6 Memory architecture (VMSA)

Unaligned data support

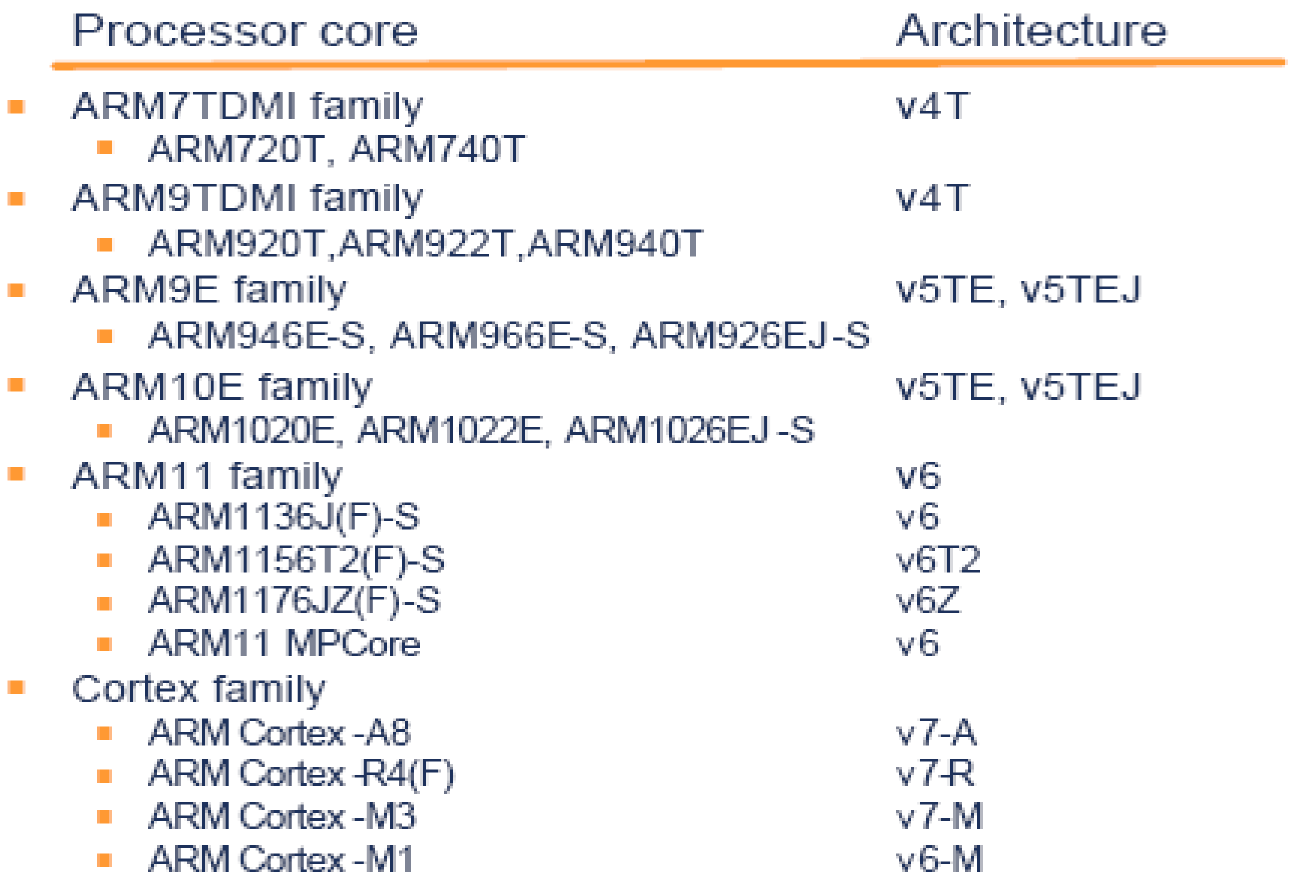
ARM1136EJ-S

ARM Cortex Processors (v7)

- ARM Cortex-**A** family (v7-A):
  - Applications processors for full OS and 3rd party applications
- ARM Cortex-**R** family (v7-R):
  - Embedded processors for real-time signal processing, control applications
- ARM Cortex-**M** family (v7-M):
  - Microcontroller-oriented processors for MCU and SoC applications



ARM Family of Processors



Processor core	Architecture
ARM7TDMI family	v4T
ARM720T, ARM740T	
ARM9TDMI family	v4T
ARM920T,ARM922T,ARM940T	
ARM9E family	v5TE, v5TEJ
ARM946E-S, ARM966E-S, ARM926EJ-S	
ARM10E family	v5TE, v5TEJ
ARM1020E, ARM1022E, ARM1026EJ-S	
ARM11 family	v6
ARM1136J(F)-S	v6
ARM1156T2(F)-S	v6T2
ARM1176JZ(F)-S	v6Z
ARM11 MPCore	v6
Cortex family	
ARM Cortex-A8	v7-A
ARM Cortex-R4(F)	v7-R
ARM Cortex-M3	v7-M
ARM Cortex-M1	v6-M

Data Sizes and Instruction Sets

- **The ARM is a 32-bit architecture.**

- **When used in relation to the ARM:**
  - **Byte** means 8 bits
  - **Halfword** means 16 bits (two bytes)
  - **Word** means 32 bits (four bytes)

- **Most ARM's implement two instruction sets**
  - 32-bit ARM Instruction Set
  - 16-bit Thumb Instruction Set

- **Jazelle cores can also execute Java bytecode**

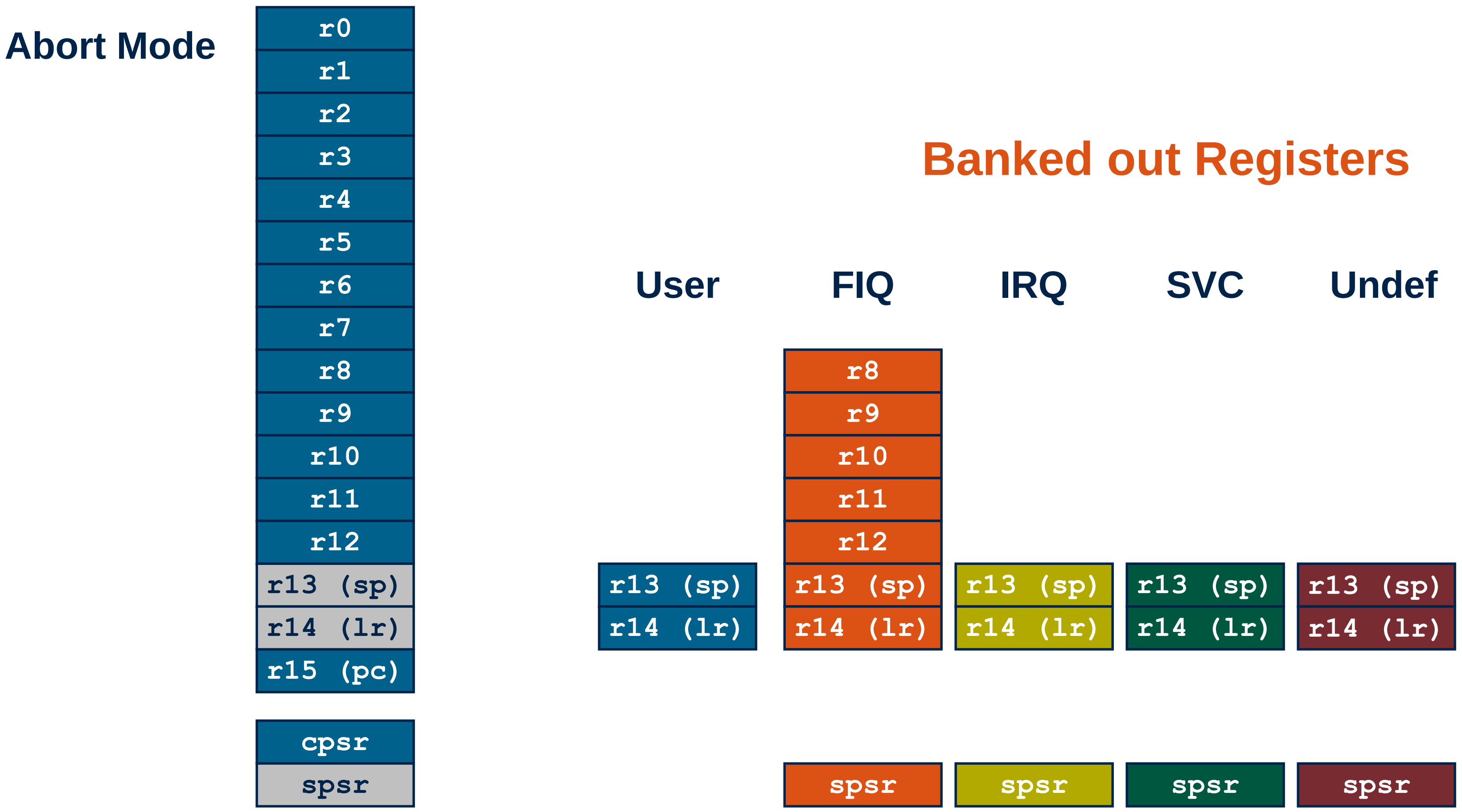
Processor Modes

- **The ARM has seven basic operating modes:**
  - **User** : unprivileged mode under which most tasks run
  - **FIQ** : entered when a high priority (fast) interrupt is raised
  - **IRQ** : entered when a low priority (normal) interrupt is raised
  - **Supervisor** : entered on reset and when a Software Interrupt instruction is executed
  - **Abort** : used to handle memory access violations
  - **Undef** : used to handle undefined instructions
  - **System** : privileged mode using the same registers as user mode

The ARM Register Set

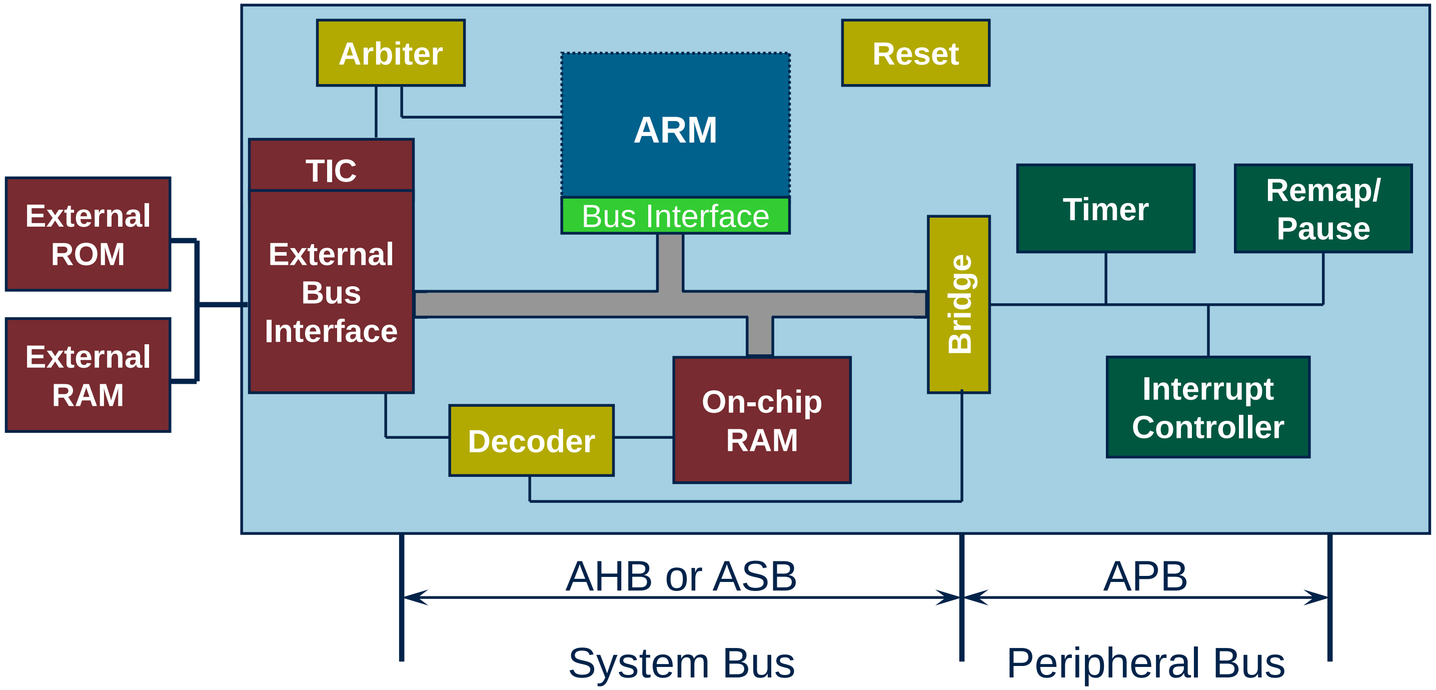
Current Visible Registers

Banked out Registers



ARM

AMBA



- **AMBA**
  - Advanced Microcontroller Bus Architecture
- **ADK**
  - Complete AMBA Design Kit
- **ACT**
  - AMBA Compliance Testbench
- **PrimeCell**
  - ARM’s AMBA compliant peripherals

Part 2

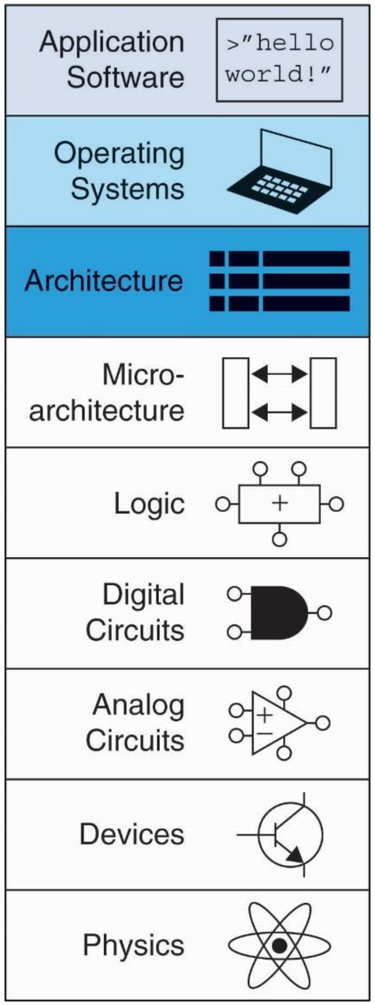
Chapter 6

***Digital Design and Computer Architecture*: ARM® Edition**

Sarah L. Harris and David Money Harris

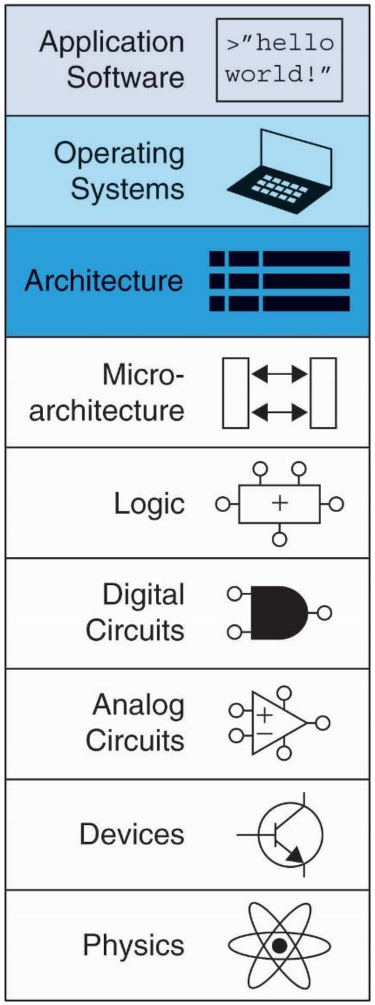
Chapter 6 :: Topics

- **Introduction**
- **Assembly Language**
- **Machine Language**
- **Programming**
- **Addressing Modes**



Introduction

- **Jumping up a few levels of abstraction**
  - **Architecture:** programmer's view of computer
    - Defined by **instructions** & **operand locations**
  - **Microarchitecture:** how to implement an architecture in hardware (covered in Chapter 7)



Instructions

- **Commands in a computer’s language**
  - **Assembly language:** human-readable format of instructions
  - **Machine language:** computer-readable format (1’s and 0’s)

Architecture Design Principles

Underlying design principles, as articulated by Hennessy and Patterson:

1. **Regularity supports design simplicity**
2. **Make the common case fast**
3. **Smaller is faster**
4. **Good design demands good compromises**

Instruction: Addition

C Code

```
a = b + c;
```

ARM Assembly Code

```
ADD a, b, c
```

- **ADD:** mnemonic – indicates operation to perform
- **b, c:** source operands
- **a:** destination operand

Instruction: Subtraction

Similar to addition - only mnemonic changes

C Code	**ARM assembly code**
`a = b - c;`	`SUB a, b, c`

- **SUB:** mnemonic
- **b, c:** source operands
- **a:** destination operand

Design Principle 1

Regularity supports design simplicity

- Consistent instruction format
- Same number of operands (two sources and one destination)
- Ease of encoding and handling in hardware

Multiple Instructions

More complex code handled by multiple ARM instructions

C Code

```
a = b + c - d;
```

ARM assembly code

```
ADD t, b, c  ; t = b + c
SUB a, t, d  ; a = t - d
```

Design Principle 2

Make the common case fast

- ARM includes only simple, commonly used instructions
- Hardware to decode and execute instructions kept simple, small, and fast
- More complex instructions (that are less common) performed using multiple simple instructions

Design Principle 2

Make the common case fast

- ARM is a **Reduced Instruction Set Computer (RISC)**, with a small number of simple instructions
- Other architectures, such as Intel's x86, are **Complex Instruction Set Computers (CISC)**

Operand Location

Physical location in computer

– Registers
– Constants (also called *immediates*)
– Memory

Operands: Registers

- ARM has 16 registers
- Registers are faster than memory
- Each register is 32 bits
- ARM is called a “32-bit architecture” because it operates on 32-bit data

Design Principle 3

Smaller is Faster

- ARM includes only a small number of registers

ARM Register Set

Name	Use
R0	Argument / return value / temporary variable
R1-R3	Argument / temporary variables
R4-R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

Operands: Registers

- **Registers:**
  - R before number, all capitals
  - Example: “R0” or “register zero” or “register R0”

Operands: Registers

- **Registers used for specific purposes:**
  - **Saved registers:** R4-R11 hold variables
  - **Temporary registers:** R0-R3 and R12, hold intermediate values
  - Discuss others later

Instructions with Registers

Revisit `ADD` instruction

C Code

ARM Assembly Code

```
; R0 = a, R1 = b, R2 = c
```

```
a = b + c
```

```
ADD R0, R1, R2
```

Generating Constants

Generating small constants using move (`MOV`):

C Code

```
//int: 32-bit signed word
int a = 23;
int b = 0x45;
```

ARM Assembly Code

```
; R0 = a, R1 = b
MOV R0, #23
MOV R1, #0x45
```

Generating Constants

Generating small constants using move (MOV):

C Code

```
//int: 32-bit signed word
int a = 23;
int b = 0x45;
```

ARM Assembly Code

```
; R0 = a, R1 = b
MOV R0, #23
MOV R1, #0x45
```

Constant must have < 8 bits of precision

Generating Constants

Generating small constants using move (MOV):

C Code

```
//int: 32-bit signed word
int a = 23;
int b = 0x45;
```

ARM Assembly Code

```
; R0 = a, R1 = b
MOV R0, #23
MOV R1, #0x45
```

Constant must have < 8 bits of precision

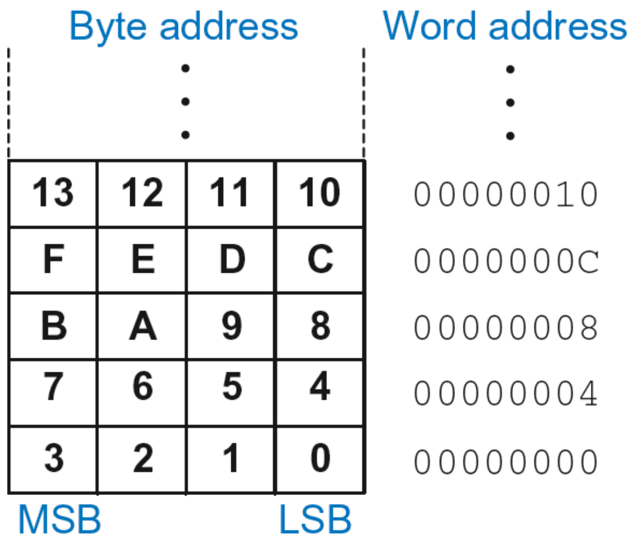
Note: `MOV` can also use 2 registers: `MOV R7, R9`

Operands: Memory

- Too much data to fit in only 16 registers
- Store more data in memory
- Memory is large, but slow
- Commonly used variables still kept in registers

Byte-Addressable Memory

- Each data **byte** has unique address
- 32-bit word = 4 bytes, so word address increments by 4



Reading Memory

- Memory read called ***load***
- **Mnemonic:** *load register* (`LDR`)
- **Format:**

```
LDR R0, [R1, #12]
```

Reading Memory

- Memory read called ***load***
- **Mnemonic:** *load register* (`LDR`)
- **Format:**

```
LDR R0, [R1, #12]
```

Address calculation:

- add *base address* (R1) to the *offset* (12)
- address = (R1 + 12)

Result:

- R0 holds the data at memory address (R1 + 12)

Writing Memory

- Memory write are called ***stores***
- **Mnemonic:** *store register* (`STR`)

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`

Writing Memory

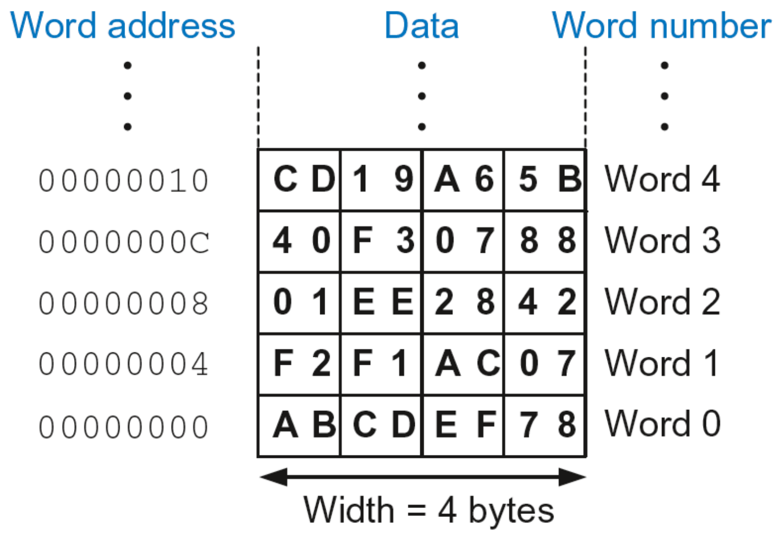
- **Example:** Store the value held in R7 into memory word 21.

Writing Memory

- **Example:** Store the value held in R7 into memory word 21.
- Memory address = 4 x 21 = 84 = 0x54

ARM assembly code

```
MOV R5, #0
STR R7, [R5, #0x54]
```



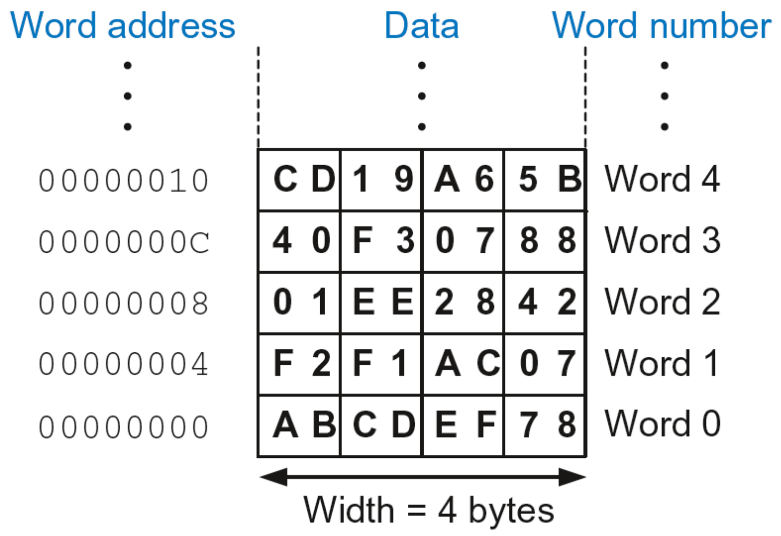
Writing Memory

- **Example:** Store the value held in R7 into memory word 21.
- Memory address = 4 x 21 = 84 = 0x54

ARM assembly code

```
MOV R5, #0
STR R7, [R5, #0x54]
```

The offset can be written in decimal or hexadecimal



Recap: Accessing Memory

- Address of a memory **word** must be multiplied by 4
- **Examples:**
  - Address of memory word 2 = 2 × 4 = 8
  - Address of memory word 10 = 10 × 4 = 40

Programming Building Blocks

- **Data-processing Instructions**
- **Conditional Execution**
- **Branches**
- **High-level Constructs:**
  - if/else statements
  - for loops
  - while loops
  - arrays
  - function calls

Programming Building Blocks

- **Data-processing Instructions**
- **Conditional Execution**
- **Branches**
- **High-level Constructs:**
  - if/else statements
  - for loops
  - while loops
  - arrays
  - function calls

Data-processing Instructions

- Logical operations
- Shifts / rotate
- Multiplication

Logical Instructions

- `AND`
- `ORR`
- `EOR` **(XOR)**
- `BIC` **(Bit Clear)**
- `MVN` **(MoVe and NOT)**

ARM

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
```

  - Comparisons set flags only - they do not specify Rd
  - Data movement does not specify Rn

- **Second operand is sent to the ALU via barrel shifter.**

Logical Instructions: Examples

Source registers

R1	0100 0110	1010 0001	1111 0001	1011 0111
R2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly code

```
AND  R3, R1, R2
ORR  R4, R1, R2
EOR  R5, R1, R2
BIC  R6, R1, R2
MVN  R7, R2
```

Result

R3	0100 0110	1010 0001	0000 0000	0000 0000
R4	1111 1111	1111 1111	1111 0001	1011 0111
R5	1011 1001	0101 1110	1111 0001	1011 0111
R6	0000 0000	0000 0000	1111 0001	1011 0111
R7	0000 0000	0000 0000	1111 1111	1111 1111

Logical Instructions: Uses

- `AND` or `BIC`: useful for **masking** bits

Example: Masking all but the least significant byte of a value

```
0xF234012F AND 0x000000FF = 0x0000002F
0xF234012F BIC 0xFFFFFF00 = 0x0000002F
```

Logical Instructions: Uses

- AND or BIC: useful for **masking** bits

  Example: Masking all but the least significant byte of a value

  0xF234012F AND 0x000000FF = 0x0000002F

  0xF234012F BIC 0xFFFFFF00 = 0x0000002F

- ORR: useful for **combining** bit fields

  Example: Combine 0xF2340000 with 0x000012BC:

  0xF2340000 ORR 0x000012BC = 0xF23412BC

Shift Instructions

- `LSL`: logical shift left

  Example: `LSL R0, R7, #5   ; R0=R7 << 5`

- `LSR`: logical shift right

  Example: `LSR R3, R2, #31  ; R3=R2 >> 31`

- `ASR`: arithmetic shift right

  Example: `ASR R9, R11, R4  ;` R9=R11 >>> $R4_{7:0}$

- `ROR`: rotate right

  Example: `ROR R8, R1, #3   ; R8=R1 ROR 3`

Shift Instructions: Example 1

- **Immediate** shift amount (5-bit immediate)
- Shift amount: 0-31

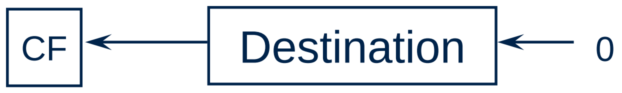
Source register

R5	1111 1111	0001 1100	0001 0000	1110 0111

Assembly Code		Result			
`LSL R0, R5, #7`	R0	1000 1110	0000 1000	0111 0011	1000 0000
`LSR R1, R5, #17`	R1	0000 0000	0000 0000	0111 1111	1000 1110
`ASR R2, R5, #3`	R2	1111 1111	1110 0011	1000 0010	0001 1100
`ROR R3, R5, #21`	R3	1110 0000	1000 0111	0011 1111	1111 1000

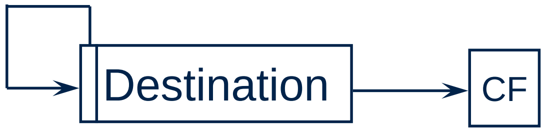
The Barrel Shifter

LSL : Logical Left Shift



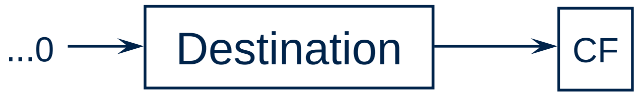
Multiplication by a power of 2

ASR: Arithmetic Right Shift



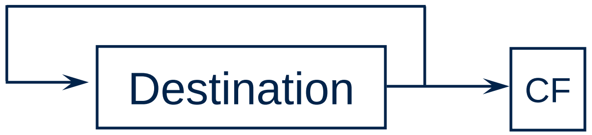
Division by a power of 2,
preserving the sign bit

LSR : Logical Shift Right



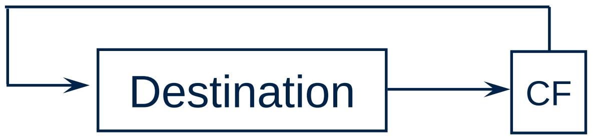
Division by a power of 2

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

Multiplication

- **`MUL`:** 32 × 32 multiplication, 32-bit result
- **`UMULL`:** Unsigned multiply long: 32 × 32 multiplication, 64-bit result
- **`SMULL`:** Signed multiply long: 32 × 32 multiplication, 64-bit result

Programming Building Blocks

- Data-processing Instructions
- **Conditional Execution**
- Branches
- High-level Constructs:
  - if/else statements
  - for loops
  - while loops
  - arrays
  - function calls

Conditional Execution

Don't always want to execute code sequentially

- For example:
  - if/else statements, while loops, etc.: only want to execute code *if* a condition is true
  - branching: jump to another portion of code *if* a condition is true
- ARM includes **condition flags** that can be:
  - set by an instruction
  - used to conditionally execute an instruction

ARM Condition Flags

Flag	Name	Description
N	**N**egative	Instruction result is negative
Z	**Z**ero	Instruction results in zero
C	**C**arry	Instruction causes an unsigned carry out
V	o**V**erflow	Instruction causes an overflow

Setting the Condition Flags: *NZCV*

- **Compare instruction: `CMP`**

Setting the Condition Flags: *NZCV*

- **Compare instruction: `CMP`**

  Example: `CMP R5, R6`

  - Performs: R5-R6
  - Does not save result
  - Sets flags. If result:

• Is 0,	*Z*=1
• Is negative,	*N*=1
• Causes a carry out,	*C*=1
• Causes a signed overflow,	*V*=1

Setting the Condition Flags: *NZCV*

- **Compare instruction: `CMP`**

  Example: `CMP R5, R6`
  - Performs: R5-R6
  - If result is 0 (Z=1), negative (N=1), etc.
  - Does not save result
- **Append instruction mnemonic with "`S`"**

Setting the Condition Flags: *NZCV*

- **Compare instruction: `CMP`**

  Example: `CMP R5, R6`
  - Performs: R5-R6
  - If result is 0 (Z=1), negative (N=1), etc.
  - Does not save result
- **Append instruction mnemonic with "S"**

  Example: `ADDS R1, R2, R3`
  - Performs: R2 + R3
  - If result is 0 (Z=1), negative (N=1), etc.
  - Saves result in R1

Condition Mnemonics

- Instruction may be ***conditionally executed*** based on the condition flags
- Condition of execution is encoded as a ***condition mnemonic*** appended to the instruction mnemonic

  Example:

  ```
  CMP    R1, R2
  SUBNE  R3, R5, R8
  ```

  - **NE**: condition mnemonic
  - `SUB` will only execute if R1 ≠ R2 (i.e., Z = 0)

Condition Mnemonics

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\bar{Z}$
0010	CS / HS	Carry set / Unsigned higher or same	C
0011	CC / LO	Carry clear / Unsigned lower	$\bar{C}$
0100	MI	Minus / Negative	N
0101	PL	Plus / Positive of zero	$\bar{N}$
0110	VS	Overflow / Overflow set	V
0111	VC	No overflow / Overflow clear	$\bar{V}$
1000	HI	Unsigned higher	$\bar{Z}C$
1001	LS	Unsigned lower or same	$Z\ OR\ \bar{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\bar{Z}(\overline{N \oplus V})$
1101	LE	Signed less than or equal	$Z\ OR\ (N \oplus V)$
1110	AL (or none)	Always / unconditional	ignored

ELSEVIER

Conditional Execution

Example:

```
CMP   R5, R9           ; performs R5-R9
                       ; sets condition flags

SUBEQ R1, R2, R3       ; executes if R5==R9 (Z=1)
ORRMI R4, R0, R9       ; executes if R5-R9 is
                       ; negative (N=1)
```

Conditional Execution

Example:

```
CMP   R5, R9        ; performs R5-R9
                    ; sets condition flags

SUBEQ R1, R2, R3    ; executes if R5==R9 (Z=1)
ORRMI R4, R0, R9    ; executes if R5-R9 is
                    ; negative (N=1)
```

Suppose R5 = 17, R9 = 23:

`CMP` performs: 17 – 23 = -6 (Sets flags: *N*=1, *Z*=0, *C*=0, *V*=0)

`SUBEQ` **doesn't execute** (they aren't equal: *Z*=0)

`ORRMI` **executes** because the result was negative (*N*=1)

ARM

Conditional Execution and Flags

- **ARM instructions can be made to execute conditionally by postfixing them with the appropriate condition code field.**
  - This improves code density *and* performance by reducing the number of forward branch instructions.

```
  CMP    r3,#0
   BEQ    skip
   ADD    r0,r1,r2
skip
```

```
CMP    r3,#0
 ADDNE r0,r1,r2
```

- **By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using “S”. CMP does not need “S”.**

```
loop
   …
   SUBS r1,r1,#1
   BNE  loop
```

decrement r1 and set flags

if Z flag clear then branch

ARM

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	

Examples of conditional execution

- Use a sequence of several conditional instructions

```
    if (a==0) func(1);
        CMP         r0,#0
        MOVEQ       r0,#1
        BLEQ        func
```

- Set the flags, then use various condition codes

```
    if (a==0) x=0;
    if (a>0)  x=1;
        CMP         r0,#0
        MOVEQ       r1,#0
        MOVGT       r1,#1
```

- Use conditional compare instructions

```
    if (a==4 || a==10) x=0;
        CMP         r0,#4
        CMPNE       r0,#10
        MOVEQ       r1,#0
```

Programming Building Blocks

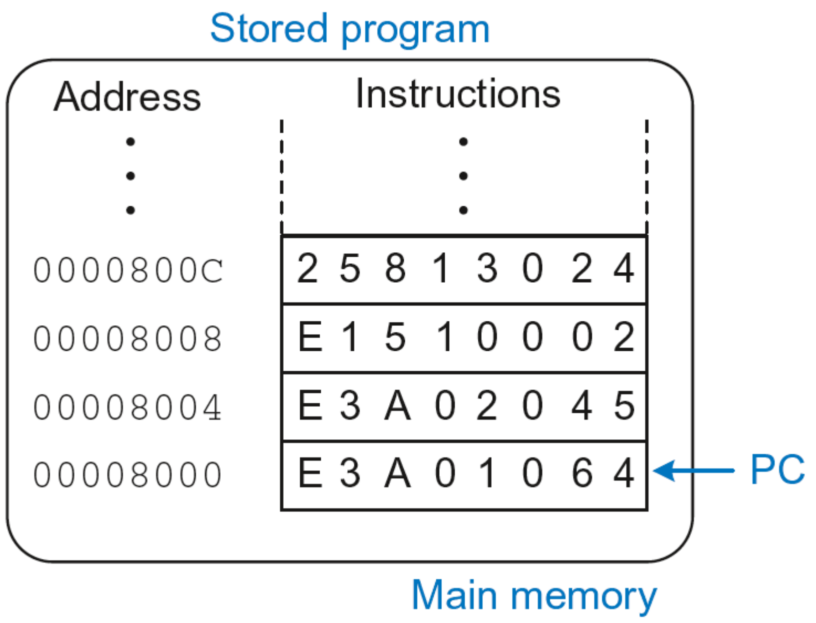
- Data-processing Instructions
- Conditional Execution
- **Branches**
- High-level Constructs:
  - if/else statements
  - for loops
  - while loops
  - arrays
  - function calls

Branching

- Branches enable out of sequence instruction execution
- Types of branches:
  - **Branch (`B`)**
    - branches to another instruction
  - **Branch and link (`BL`)**
    - discussed later
- Both can be conditional or unconditional

The Stored Program

Assembly code	Machine code
`MOV   R1, #100`	`0xE3A01064`
`MOV   R2, #69`	`0xE3A02045`
`CMP   R1, R2`	`0xE1510002`
`STRHS R3, [R1, #0x24]`	`0x25813024`



Unconditional Branching (B)

ARM assembly

```
   MOV R2, #17           ; R2 = 17
   B   TARGET            ; branch to target
   ORR R1, R1, #0x4      ; not executed

TARGET
       SUB R1, R1, #78       ; R1 = R1 + 78
```

Labels (like TARGET) indicate instruction location. Labels can't be reserved words (like ADD, ORR, etc.)

The Branch Not Taken

ARM Assembly

```
   MOV  R0, #4          ; R0 = 4
   ADD  R1, R0, R0      ; R1 = R0+R0 = 8
   CMP  R0, R1          ; sets flags with R0-R1
   BEQ  THERE           ; branch not taken (Z=0)
   ORR  R1, R1, #1      ; R1 = R1 OR R1 = 9
THERE
   ADD R1, R1, 78       ; R1 = R1 + 78 = 87
```

Programming Building Blocks

- Data-processing Instructions
- Conditional Execution
- Branches
- **High-level Constructs:**
  - **if/else statements**
  - **for loops**
  - **while loops**
  - arrays
  - function calls

if Statement

C Code

```
if (i == j)
  f = g + h;



f = f - i;
```

if Statement

C Code

```
if (i == j)
   f = g + h;

f = f - i;
```

ARM Assembly Code

```
;R0=f, R1=g, R2=h, R3=i, R4=j

 CMP R3, R4        ; set flags with R3-R4
 BNE L1            ; if i!=j, skip if block
 ADD R0, R1, R2    ; f = g + h

L1
 SUB R0, R0, R2    ; f = f - i
```

if Statement

C Code

```
if (i == j)
  f = g + h;

f = f - i;
```

ARM Assembly Code

```
;R0=f, R1=g, R2=h, R3=i, R4=j

 CMP R3, R4        ; set flags with R3-R4
 BNE L1            ; if i!=j, skip if block
 ADD R0, R1, R2    ; f = g + h

L1
 SUB R0, R0, R2    ; f = f - i
```

Assembly tests opposite case (`i != j`) of high-level code (`i == j`)

ELSEVIER

while Loops

C Code

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x   = 0;


while (pow != 128) {

  pow = pow * 2;
  x = x + 1;
}
```

ARM Assembly Code

```
; R0 = pow, R1 = x
  MOV   R0, #1              ; pow = 1
  MOV   R1, #0              ; x = 0

WHILE
  CMP R0, #128              ; R0-128
  BEQ DONE                  ; if (pow==128)
                            ; exit loop
  LSL R0, R0, #1            ; pow=pow*2
  ADD R1, R1, #1            ; x=x+1
  B   WHILE                 ; repeat loop

DONE
```

while Loops

C Code

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x   = 0;



while (pow != 128) {

  pow = pow * 2;
  x = x + 1;
}
```

ARM Assembly Code

```
; R0 = pow, R1 = x
  MOV   R0, #1            ; pow = 1
  MOV   R1, #0            ; x = 0

WHILE
  CMP R0, #128            ; R0-128
  BEQ DONE                ; if (pow==128)
                          ; exit loop
  LSL R0, R0, #1          ; pow=pow*2
  ADD R1, R1, #1          ; x=x+1
  B   WHILE               ; repeat loop

DONE
```

Assembly tests for the opposite case (`pow == 128`) of the C code (`pow != 128`).

for Loops

```
for (initialization; condition; loop operation)
   statement
```

- `initialization:` executes before the loop begins
- `condition:` is tested at the beginning of each iteration
- `loop operation:` executes at the end of each iteration
- `statement:` executes each time the condition is met

for Loops

C Code

```
// adds numbers from 1-9
int sum = 0


for (i=1; i!=10; i=i+1)
   sum = sum + i;
```

ARM Assembly Code

for Loops

C Code

```
// adds numbers from 1-9
int sum = 0


for (i=1; i!=10; i=i+1)
   sum = sum + i;
```

ARM Assembly Code

```
; R0 = i, R1 = sum
  MOV   R0, #1              ; i = 1
  MOV   R1, #0              ; sum = 0

FOR
  CMP R0, #10               ; R0-10
  BEQ DONE                  ; if (i==10)
                            ; exit loop
  ADD R1, R1, R0            ; sum=sum + i
  ADD R0, R0, #1            ; i = i + 1
  B   FOR                   ; repeat loop

DONE
```

Programming Building Blocks

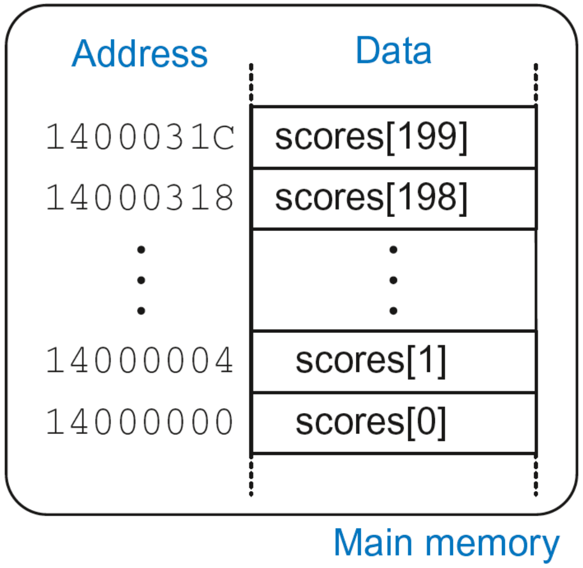
- **Data-processing Instructions**
- **Conditional Execution**
- **Branches**
- **High-level Constructs:**
  - if/else statements
  - for loops
  - while loops
  - **arrays**
  - function calls

Arrays

- Access large amounts of similar data
  - **Index:** access to each element
  - **Size:** number of elements

Arrays

- 5-element array
  - **Base address** = 0x14000000 (address of first element, scores[0])
  - Array elements accessed relative to base address



Accessing Arrays

C Code

```
int array[5];
array[0] = array[0] * 8;
array[1] = array[1] * 8;
```

ARM Assembly Code

```
; R0 = array base address
```

Accessing Arrays

C Code

```
int array[5];
array[0] = array[0] * 8;
array[1] = array[1] * 8;
```

ARM Assembly Code

```
; R0 = array base address
  MOV R0, #0x60000000          ; R0 = 0x60000000

  LDR R1, [R0]                 ; R1 = array[0]
  LSL R1, R1, 3                ; R1 = R1 << 3 = R1*8
  STR R1, [R0]                 ; array[0] = R1

  LDR R1, [R0, #4]             ; R1 = array[1]
  LSL R1, R1, 3                ; R1 = R1 << 3 = R1*8
  STR R1, [R0, #4]             ; array[1] = R1
```

Arrays using for Loops

C Code

```
int array[200];
int i;

for (i=199; i >= 0; i = i - 1)
     array[i] = array[i] * 8;
```

ARM Assembly Code

```
; R0 = array base address, R1 = i
```

Arrays using for Loops

C Code

```
int array[200];
int i;

for (i=199; i >= 0; i = i - 1)
     array[i] = array[i] * 8;
```

ARM Assembly Code

```
; R0 = array base address, R1 = i
  MOV R0, 0x60000000
  MOV R1, #199

FOR
  LDR  R2, [R0, R1, LSL #2]     ; R2 = array(i)
  LSL  R2, R2, #3               ; R2 = R2<<3 = R3*8
  STR  R2, [R0, R1, LSL #2]     ; array(i) = R2
  SUBS R0, R0, #1               ; i = i - 1
                                ; and set flags
  BPL  FOR                      ; if (i>=0) repeat loop
```

Programming Building Blocks

- Data-processing Instructions
- Conditional Execution
- Branches
- **High-level Constructs:**
  - if/else statements
  - for loops
  - while loops
  - arrays
  - **function calls**

Function Calls

- **Caller:** calling function (in this case, `main`)
- **Callee:** called function (in this case, `sum`)

C Code

```
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

int sum(int a, int b)
{
  return (a + b);
}
```

Function Conventions

- **Caller:**
  - passes **arguments** to callee
  - jumps to callee

Function Conventions

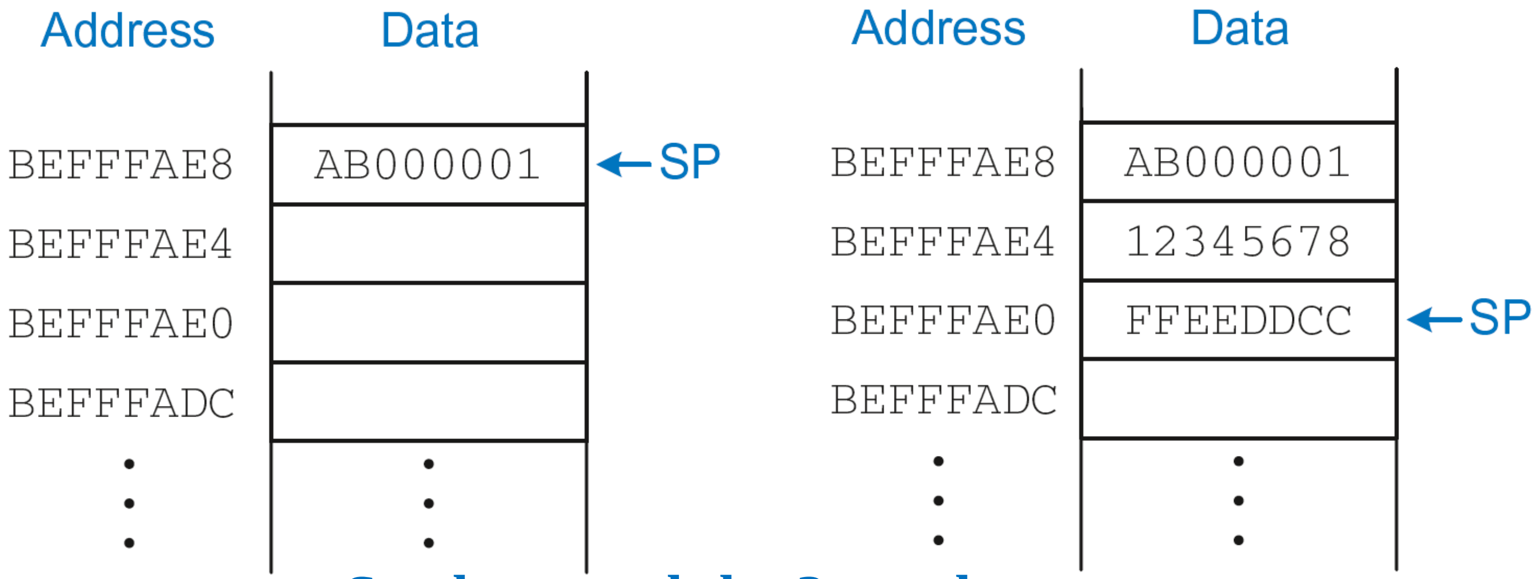
- **Caller:**
  - passes **arguments** to callee
  - jumps to callee
- **Callee:**
  - **performs** the function
  - **returns** result to caller
  - **returns** to point of call
  - **must not overwrite** registers or memory needed by caller

The Stack

- Memory used to temporarily save variables
- Like stack of dishes, last-in-first-out (LIFO) queue
- ***Expands***: uses more memory when more space needed
- ***Contracts***: uses less memory when the space no longer needed

The Stack

- Grows down (from higher to lower memory addresses)
- Stack pointer: `SP` points to top of the stack



Stack expands by 3 words

How to Encode Instructions?

- **Design Principle 1: Regularity supports design simplicity**
  - 32-bit data, 32-bit instructions
  - For design simplicity, would prefer a single instruction format but…

How to Encode Instructions?

- **Design Principle 1: Regularity supports design simplicity**
  - 32-bit data, 32-bit instructions
  - For design simplicity, would prefer a single instruction format but…
  - Instructions have different needs

Design Principle 4

Good design demands good compromises

- Multiple instruction formats allow flexibility
  - `ADD`, `SUB`: use 3 register operands
  - `LDR`, `STR`: use 2 register operands and a constant
- Number of instruction formats kept small
  - to adhere to design principles 1 and 3 (regularity supports design simplicity and smaller is faster)

Machine Language

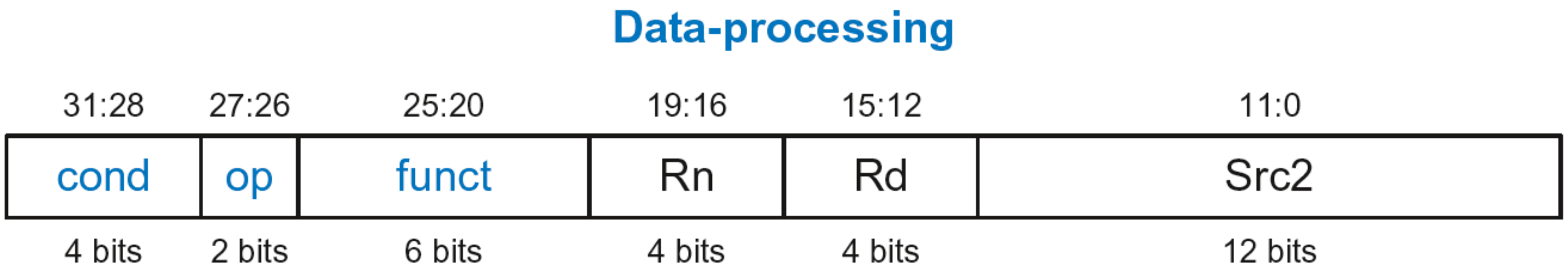
- **Binary representation of instructions**
- Computers only understand **1's and 0's**
- **32-bit instructions**
  - Simplicity favors regularity: 32-bit data & instructions
- **3 instruction formats:**
  - Data-processing
  - Memory
  - Branch

Instruction Formats

- **Data-processing**
- Memory
- Branch

Data-processing Instruction Format

- **Operands:**
  - *Rn*: first source register
  - *Src2*: second source – register or immediate
  - *Rd*: destination register
- **Control fields:**
  - *cond*: specifies conditional execution
  - *op*: the *operation code* or *opcode*
  - *funct*: the *function*/operation to perform

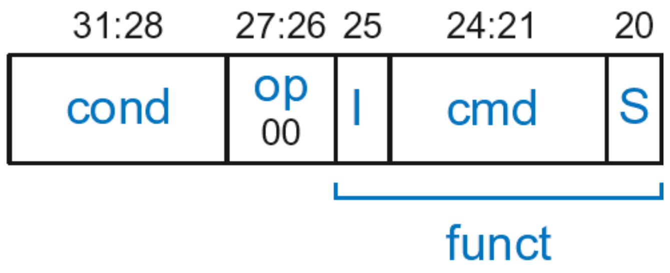


Data-processing

31:28	27:26	25:20	19:16	15:12	11:0
cond	op	funct	Rn	Rd	Src2
4 bits	2 bits	6 bits	4 bits	4 bits	12 bits

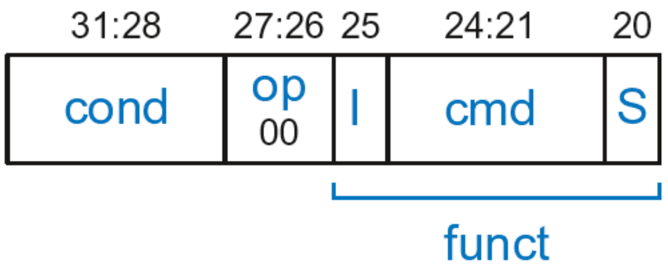
Data-processing Control Fields

- ***op*** **= 00_2** for data-processing (DP) instructions
- ***funct*** is composed of ***cmd***, ***I***-bit, and ***S***-bit



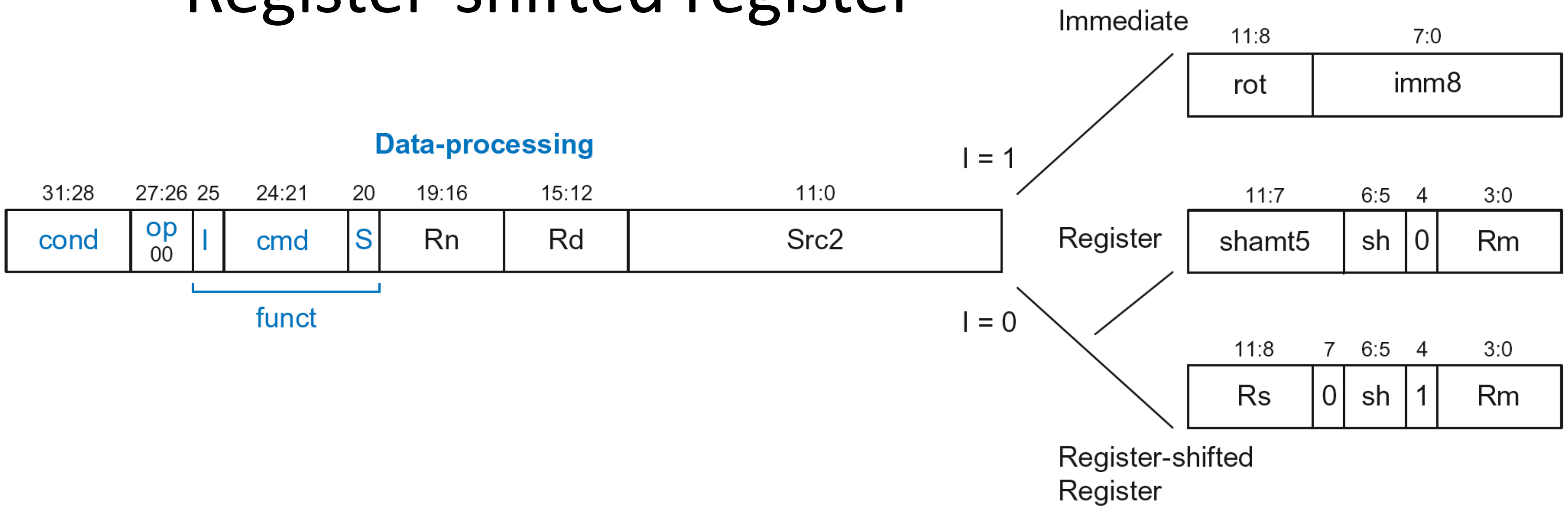
Data-processing Control Fields

- ***op*** **= 00$_2$** for data-processing (DP) instructions
- ***funct*** is composed of ***cmd***, ***I***-bit, and ***S***-bit
  - ***cmd:*** specifies the specific data-processing instruction. For example,
    - ***cmd*** = 0100$_2$ for `ADD`
    - ***cmd*** = 0010$_2$ for `SUB`
  - ***I***-bit
    - ***I*** = 0: *Src2* is a register
    - ***I*** = 1: *Src2* is an immediate
  - ***S***-bit: 1 if sets condition flags
    - ***S*** = 0: `SUB  R0, R5, R7`
    - ***S*** = 1: `ADDS R8, R2, R4` or `CMP R3, #10`



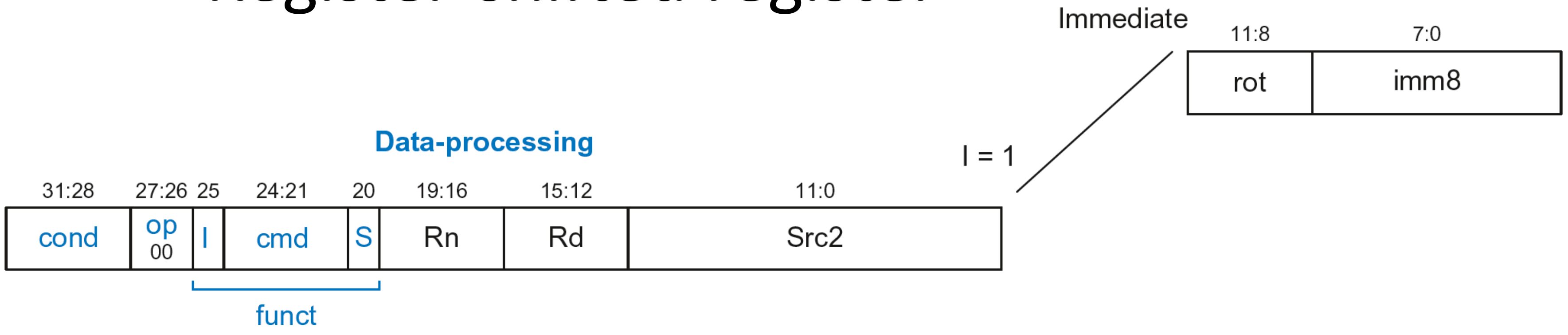
Data-processing *Src2* Variations

- *Src2* can be:
  - Immediate
  - Register
  - Register-shifted register



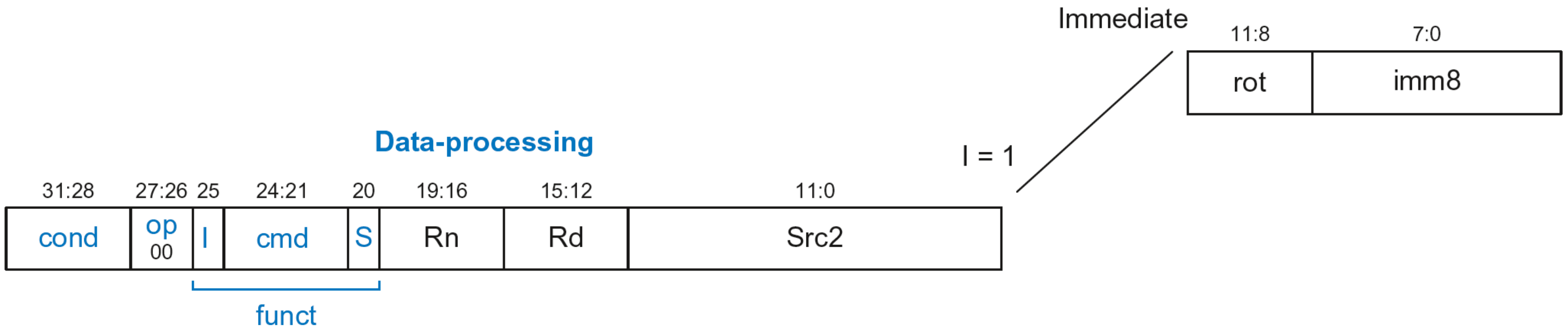
Data-processing *Src2* Variations

- *Src2* can be:
  - **Immediate**
  - Register
  - Register-shifted register



Immediate *Src2*

- **Immediate encoded as:**
  - *imm8*: 8-bit unsigned immediate
  - *rot*: 4-bit rotation value
- **32-bit constant is:** *imm8* **ROR** (*rot* × 2)



Immediate *Src2*

- **Immediate encoded as:**
  - *imm8*: 8-bit unsigned immediate
  - *rot*: 4-bit rotation value
- **32-bit constant is:** *imm8* **ROR** (*rot* × 2)
- **Example:** *imm8* = abcdefg

rot	**32-bit constant**
0000	0000 0000 0000 0000 0000 0000 abcd efgh
0001	gh00 0000 0000 0000 0000 0000 00ab cdef
…	…
1111	0000 0000 0000 0000 0000 00ab cdef gh00

Immediate *Src2*

- **Immediate encoded as:**
  - *imm8*: 8-bit unsigned immediate
  - *rot*: 4-bit rotation value
- **32-bit constant is:** *imm8* **ROR** (*rot* × 2)
- **Example:** *imm8* = abcdefg

ROR by X = ROL by (32-X)
Ex: ROR by 30 = ROL by 2

rot	32-bit constant
0000	0000 0000 0000 0000 0000 0000 abcd efgh
0001	gh00 0000 0000 0000 0000 0000 00ab cdef
…	…
1111	0000 0000 0000 0000 0000 00ab cdef gh00

DP Instruction with Immediate *Src2*

```
ADD R0, R1, #42
```

- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 0100_2 (4) for ADD
- ***Src2*** is an immediate so ***I*** = 1
- ***Rd*** = 0, ***Rn*** = 1
- ***imm8*** = 42, ***rot*** = 0

DP Instruction with Immediate *Src2*

```
ADD R0, R1, #42
```

- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 0100_2 (4) for ADD
- ***Src2*** is an immediate so ***I*** = 1
- ***Rd*** = 0, ***Rn*** = 1
- ***imm8*** = 42, ***rot*** = 0

Field Values

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
1110_2	00_2	1	0100_2	0	1	0	0	42
cond	op	I	cmd	S	Rn	Rd	shamt5	sh Rm
1110	00	1	0100	0	0001	0000	0000	00101010

DP Instruction with Immediate *Src2*

```
ADD R0, R1, #42
```

- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 0100_2 (4) for ADD
- ***Src2*** is an immediate so ***I*** = 1
- ***Rd*** = 0, ***Rn*** = 1
- ***imm8*** = 42, ***rot*** = 0

Field Values

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
1110_2	00_2	1	0100_2	0	1	0	0	42
cond	op	I	cmd	S	Rn	Rd	shamt5	sh Rm
1110	00	1	0100	0	0001	0000	0000	00101010

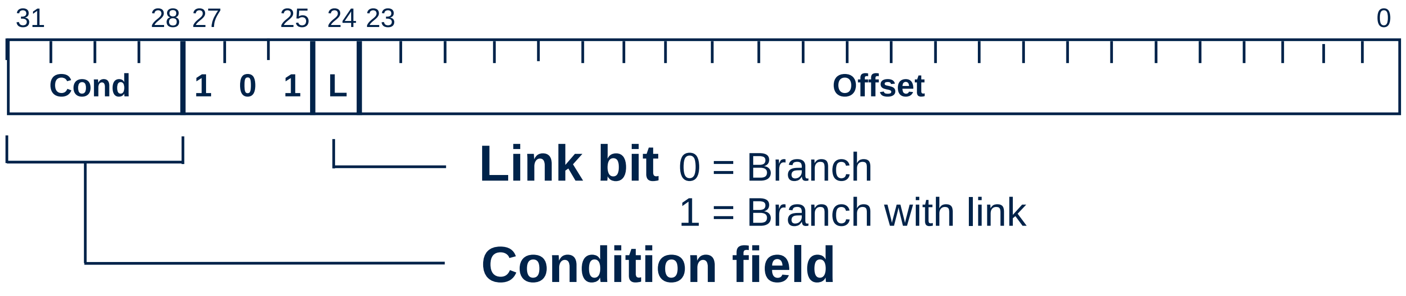
0xE281002A

Shift Instructions Encoding

Shift Type	*sh*
LSL	00_2
LSR	01_2
ASR	10_2
ROR	11_2

Branch instructions

- **Branch :** `B{<cond>} label`
- **Branch with Link :** `BL{<cond>} subroutine_label`



- **The processor core shifts the offset field left by 2 positions, sign-extends it and adds it to the PC**
  - ± 32 Mbyte range
  - How to perform longer branches?

ARM®

Multiply

- **Syntax:**

MUL{<cond>}{S} Rd, Rm, Rs	Rd = Rm * Rs
MLA{<cond>}{S} Rd,Rm,Rs,Rn	Rd = (Rm * Rs) + Rn
[U|S]MULL{<cond>}{S} RdLo, RdHi, Rm, Rs	RdHi,RdLo := Rm*Rs
[U|S]MLAL{<cond>}{S} RdLo, RdHi, Rm, Rs	RdHi,RdLo := (Rm*Rs)+RdHi,RdLo

- **Cycle time**
  - Basic MUL instruction
    - 2-5 cycles on ARM7TDMI
    - 1-3 cycles on StrongARM/XScale
    - 2 cycles on ARM9E/ARM102xE
  - +1 cycle for ARM9TDMI (over ARM7TDMI)
  - +1 cycle for accumulate (not on 9E though result delay is one cycle longer)
  - +1 cycle for “long”

- **Above are “general rules” - refer to the TRM for the core you are using for the exact details**

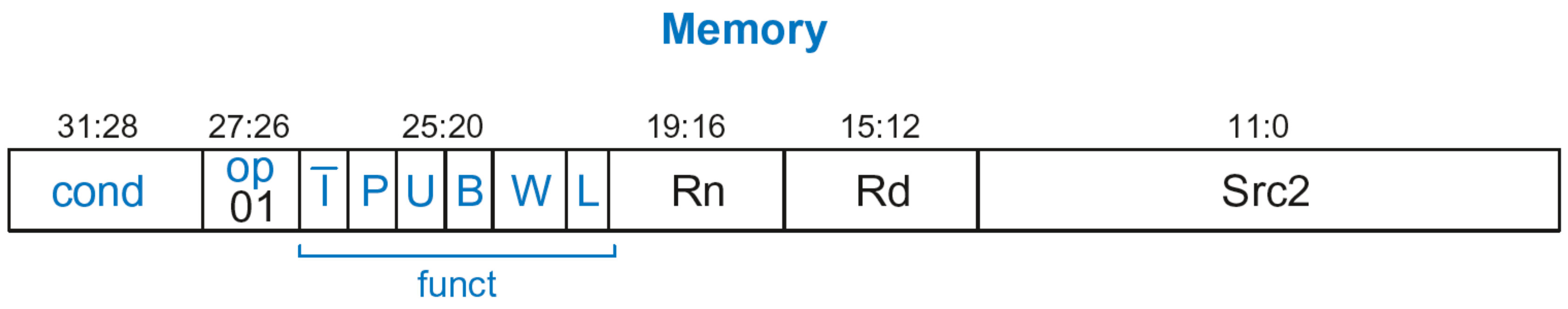
Instruction Formats

- Data-processing
- **Memory**
- Branch

Memory Instruction Format

Encodes: `LDR`, `STR`, `LDRB`, `STRB`

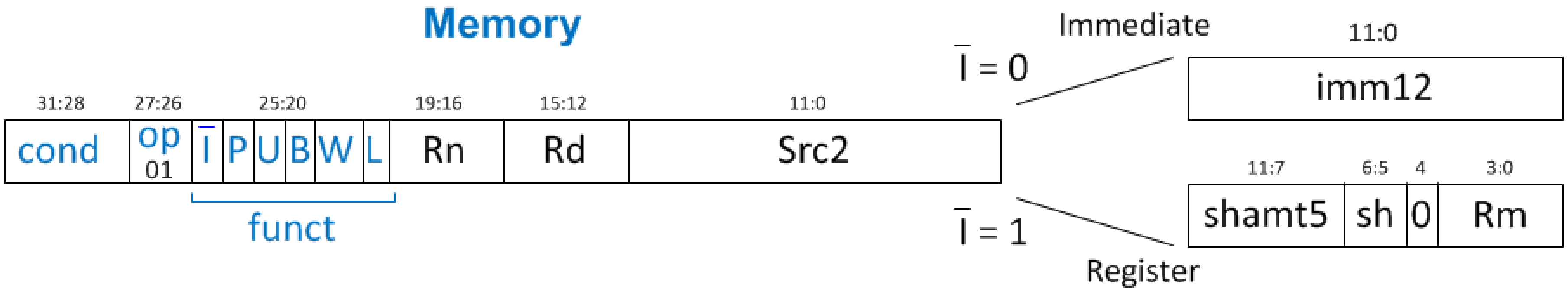
• ***op*** =	01_2
• ***Rn*** =	base register
• ***Rd*** =	destination (load), source (store)
• ***Src2*** =	offset
• ***funct*** =	6 control bits



Review: Memory Instruction Format

Encodes: `LDR, STR, LDRB, STRB`

- ***op*** = 01_2
- ***Rn*** = base register
- ***Rd*** = destination (load), source (store)
- ***Src2*** = offset: immediate or register (optionally shifted)
- ***funct*** = $\overline{I}$ (immediate bar), *P* (preindex), *U* (add), *B* (byte), *W* (writeback), *L* (load)



Offset Options

Recall: Address = Base Address + Offset

Example: `LDR R1, [R2, #4]`

Base Address = R2, Offset = 4

Address = (R2 + 4)

- Base address always in a register
- The offset can be:
  - an immediate
  - a register
  - or a scaled (shifted) register

Offset Examples

ARM Assembly	Memory Address
`LDR R0, [R3, #4]`	R3 + 4
`LDR R0, [R5, #-16]`	R5 – 16
`LDR R1, [R6, R7]`	R6 + R7
`LDR R2, [R8, -R9]`	R8 – R9
`LDR R3, [R10, R11, LSL #2]`	R10 + (R11 << 2)
`LDR R4, [R1, -R12, ASR #4]`	R1 – (R12 >>> 4)
`LDR R0, [R9]`	R9

Indexing Modes

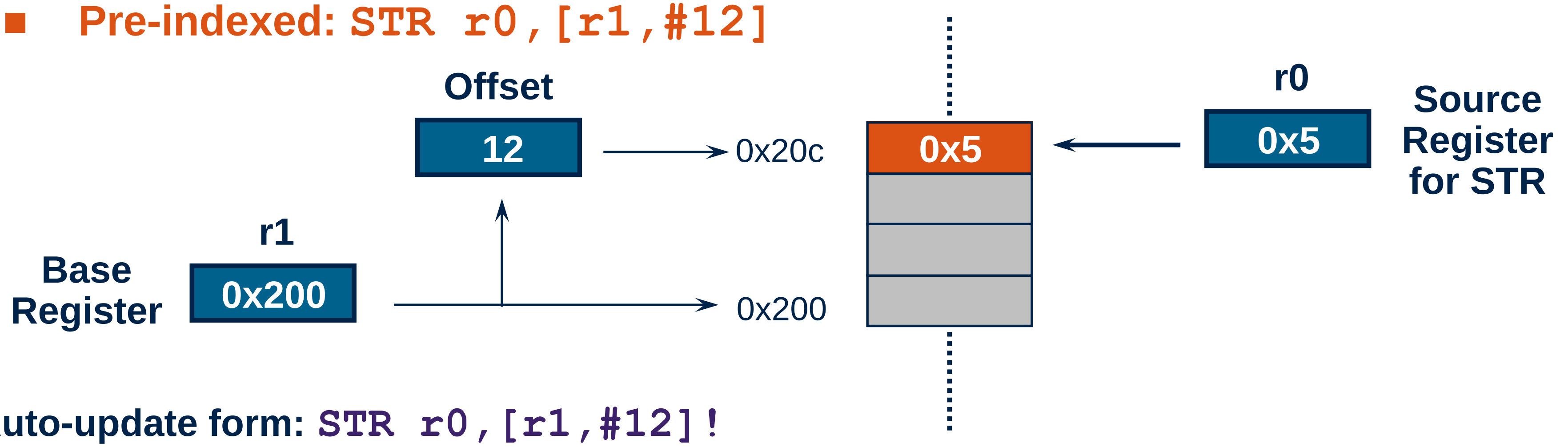
Mode	Address	Base Reg. Update
Offset	Base register ± Offset	No change
Preindex	Base register ± Offset	Base register ± Offset
Postindex	Base register	Base register ± Offset

Examples

- **Offset:**
```
LDR R1, [R2, #4]      ; R1 = mem[R2+4]
```
- **Preindex:**
```
LDR R3, [R5, #16]!    ; R3 = mem[R5+16]
                      ; R5 = R5 + 16
```
- **Postindex:**
```
LDR R8, [R1], #8      ; R8 = mem[R1]
                      ; R1 = R1 + 8
```

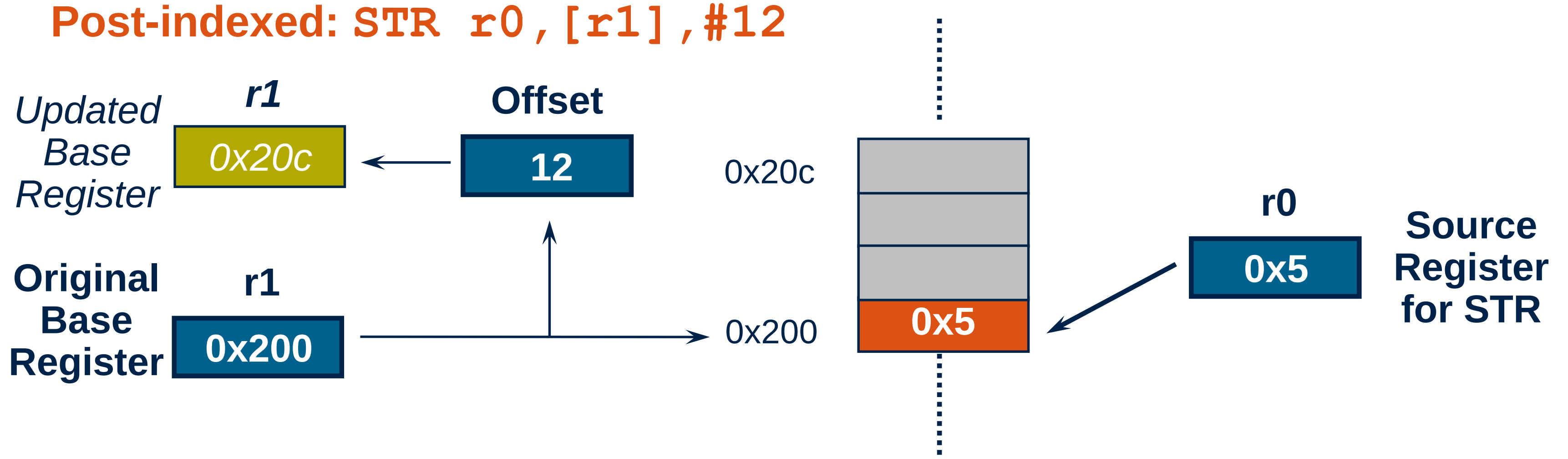
Pre or Post Indexed Addressing?

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



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

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

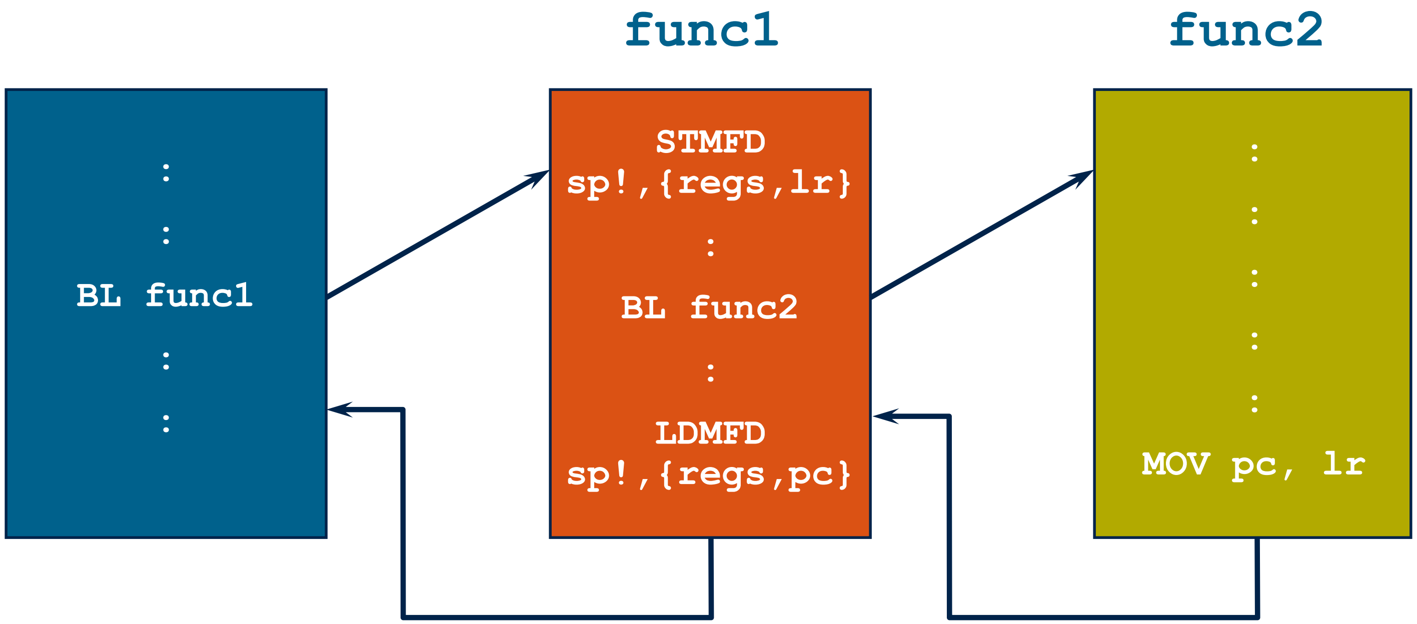


Instruction Formats

- Data-processing
- Memory
- **Branch**

ARM Branches and Subroutines

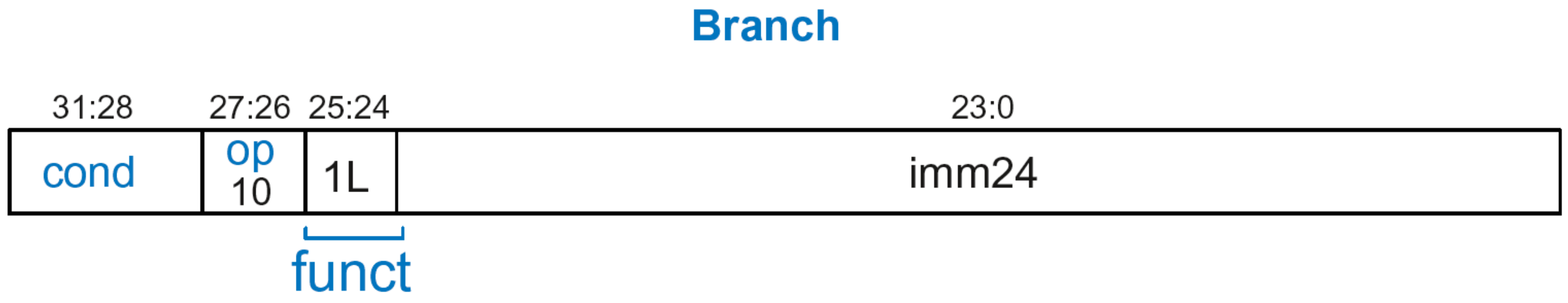
- **B <label>**
  - PC relative. ±32 Mbyte range.
- **BL <subroutine>**
  - Stores return address in LR
  - Returning implemented by restoring the PC from LR
  - For non-leaf functions, LR will have to be stacked



Branch Instruction Format

Encodes `B` and `BL`

- ***op*** = 10_2
- ***imm24***: 24-bit immediate
- ***funct*** = $1L_2$: L = 1 for `BL`, L = 0 for `B`



Encoding Branch Target Address

- ***Branch Target Address* (BTA)*:*** Next PC when branch taken
- BTA is relative to current PC + 8
- *imm24* encodes BTA
- ***imm24*** = # of words BTA is away from PC+8

Branch Instruction: Example 1

ARM assembly code

```
0xA0         BLT THERE       ← PC
0xA4         ADD R0, R1, R2
0xA8         SUB R0, R0, R9  ← PC+8
0xAC         ADD SP, SP, #8
0xB0         MOV PC, LR
0xB4  THERE  SUB R0, R0, #1  ← BTA
0xB8         BL  TEST
```

- PC = 0xA0
- PC + 8 = 0xA8
- `THERE` label is 3 instructions past PC+8
- So, *imm24* = 3

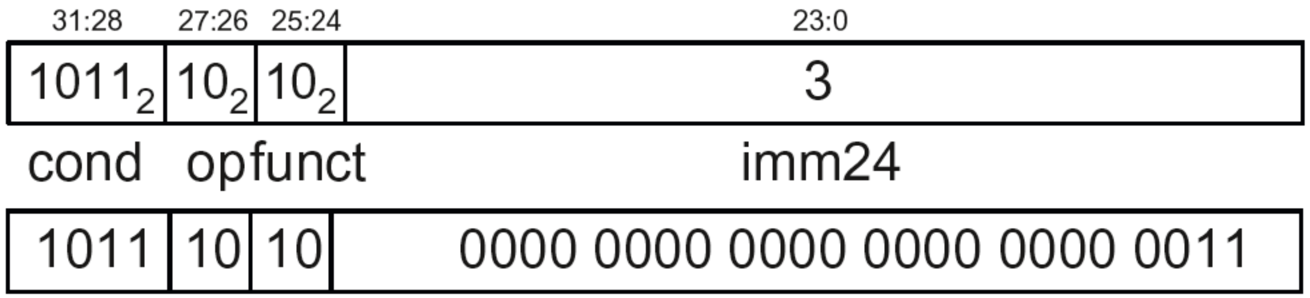
Branch Instruction: Example 1

ARM assembly code

```
0xA0         BLT  THERE         ← PC
0xA4         ADD  R0, R1, R2
0xA8         SUB  R0, R0, R9    ← PC+8
0xAC         ADD  SP, SP, #8
0xB0         MOV  PC, LR
0xB4  THERE  SUB  R0, R0, #1    ← BTA
0xB8         BL   TEST
```

- PC = 0xA0
- PC + 8 = 0xA8
- `THERE` label is 3 instructions past PC+8, so *imm24* = 3

Field Values

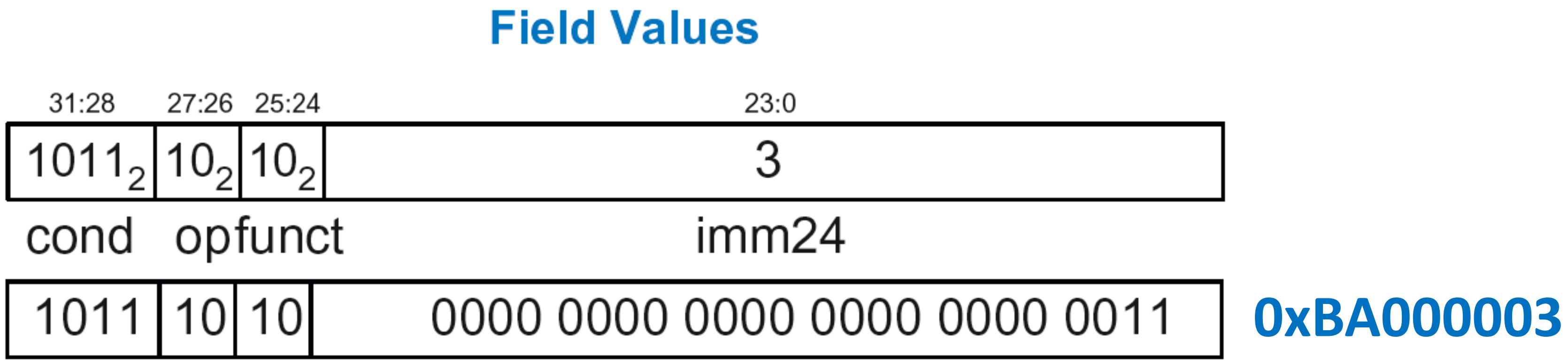


Branch Instruction: Example 1

ARM assembly code

```
0xA0          BLT  THERE      ← PC
0xA4          ADD  R0, R1, R2
0xA8          SUB  R0, R0, R9 ← PC+8
0xAC          ADD  SP, SP, #8
0xB0          MOV  PC, LR
0xB4  THERE   SUB  R0, R0, #1 ← BTA
0xB8          BL   TEST
```

- PC = 0xA0
- PC + 8 = 0xA8
- `THERE` label is 3 instructions past PC+8, so *imm24* = 3



Addressing Modes

How do we address operands?

- Register
- Immediate
- Base
- PC-Relative

Addressing Modes

How do we address operands?

- **Register Only**
- Immediate
- Base
- PC-Relative

ELSEVIER

Register Addressing

- Source and destination operands found in registers
- Used by data-processing instructions
- **Three submodes:**
  - Register-only
  - Immediate-shifted register
  - Register-shifted register

Register Addressing Examples

- **Register-only**

  Example: `ADD R0, R2, R7`

- **Immediate-shifted register**

  Example: `ORR R5, R1, R3, LSL #1`

- **Register-shifted register**

  Example: `SUB R12, R9, R0, ASR R1`

Addressing Modes

How do we address operands?

- Register Only
- **Immediate**
- Base
- PC-Relative

Immediate Addressing

- Source and destination operands found in registers and immediates

  Example: `ADD R9, R1, #14`

- Uses data-processing format with *I*=1
- Immediate is encoded as
  - 8-bit immediate (*imm8*)
  - 4-bit rotation (*rot*)
- 32-bit immediate = *imm8* ROR (*rot* x 2)

Addressing Modes

How do we address operands?

- Register Only
- Immediate
- **Base**
- PC-Relative

Base Addressing

- Address of operand is:

  base register + offset

- Offset can be a:
  - 12-bit Immediate
  - Register
  - Immediate-shifted Register

Base Addressing Examples

- **Immediate offset**

  Example: `LDR R0, [R8, #-11]`

  (R0 = mem[R8 - 11])

- **Register offset**

  Example: `LDR R1, [R7, R9]`

  (R1 = mem[R7 + R9])

- **Immediate-shifted register offset**

  Example: `STR R5, [R3, R2, LSL #4]`

  (R5 = mem[R3 + (R2 << 4)])

Addressing Modes

How do we address operands?

- Register Only
- Immediate
- Base
- **PC-Relative**

PC-Relative Addressing

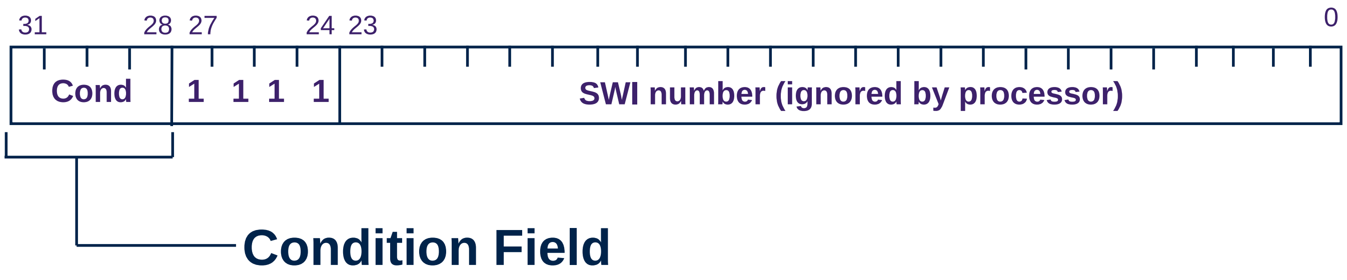
- Used for branches
- Branch instruction format:
  - Operands are PC and a signed 24-bit immediate (*imm24*)
  - Changes the PC
  - New PC is relative to the old PC
  - *imm24* indicates the number of words away from PC+8
- PC = (PC+8) + (SignExtended(*imm24*) x 4)

ARM®

Address accessed

- **Address accessed by LDR/STR is specified by a base register plus an offset**
- **For word and unsigned byte accesses, offset can be**
  - An unsigned 12-bit immediate value (ie 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]
  LDR r0,[r1,-r2,LSL#2]
  ```
- **For halfword and signed halfword / byte, offset can be:**
  - An unsigned 8 bit immediate value (ie 0-255 bytes).
  - A register (unshifted).
- **Choice of *pre-indexed* or *post-indexed* addressing**

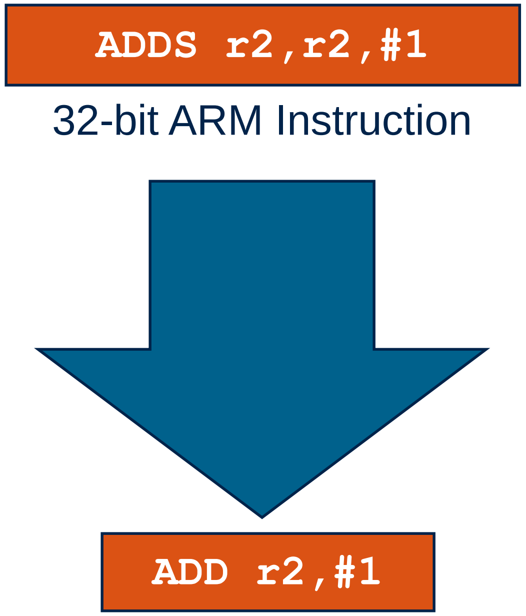
Software Interrupt (SWI)



- 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>`

Thumb

- **Thumb is a 16-bit instruction set**
  - Optimised for code density from C code (~65% of ARM code size)
  - Improved performance from narrow memory
  - Subset of the functionality of the ARM instruction set
- **Core has additional execution state - Thumb**
  - Switch between ARM and Thumb using **`BX`** instruction



32-bit ARM Instruction

16-bit Thumb Instruction

For most instructions generated by compiler:

- Conditional execution is not used
- Source and destination registers identical
- Only Low registers used
- Constants are of limited size
- Inline barrel shifter not used

Interpreting Machine Code

- **Start with *op*:** tells how to parse rest

  op = 00 (Data-processing)

  op = 01 (Memory)

  op = 10 (Branch)

- ***I*-bit:** tells how to parse ***Src2***
- **Data-processing instructions:**

  If ***I***-bit is 0, bit 4 determines if ***Src2*** is a register (bit 4 = 0) or a register-shifted register (bit 4 = 1)

- **Memory instructions:**

  Examine ***funct*** bits for indexing mode, instruction, and add or subtract offset

Interpreting Machine Code: Example 1

0xE0475001

Interpreting Machine Code: Example 1

0xE0475001

- **Start with *op*:** 00_2, so data-processing instruction
- ***cmd*:** 0010_2 (2), so SUB
- ***I*-bit:** 0, so *Src2* is a register
- **bit 4:** 0, so *Src2* is a register (optionally shifted by *shamt5*)
- **Rn**=7, **Rd**=5, **Rm**=1, ***shamt5*** = 0, ***sh*** = 0
- So, instruction is: **`SUB R5,R7,R1`**

Up Next

How to implement the ARM Instruction Set Architecture in Hardware

Microarchitecture