TURIN POLYTECHNIC UNIVERSITY IN TASHKENT

Fundamentals of Computer Architecture

ARM – Data Processing Instructions, Conditional Execution

Teacher: Lobar Asretdinova

Table 2, General-purpose registers and AAPCS64 usage

Register	Special	Role in the procedure call standard
SP		The Stack Pointer.
r30	LR	The Link Register.
r29	FP	The Frame Pointer
r19…r28		Callee-saved registers
r18		The Platform Register, if needed; otherwise a temporary register. See notes.
r17	IP1	The second intra-procedure-call temporary register (can be used by call veneers and PLT code); at other times may be used as a temporary register.
r16	IP0	The first intra-procedure-call scratch register (can be used by call veneers and PLT code); at other times may be used as a temporary register.
r9…r15		Temporary registers
r8		Indirect result location register
r0…r7		Parameter/result registers

Type	Mnemonic	Instruction
Arithmetic Register	**ADD**	Add
	ADDS	Add and set flags
	SUB	Subtract
	SUBS	Subtract and set flags
	CMP	Compare
	CMN	Compare negative
	NEG	Negate
	NEGS	Negate and set flags
Arithmetic Immediate	**ADDI**	Add Immediate
	ADDIS	Add and set flags Immediate
	SUBI	Subtract Immediate
	SUBIS	Subtract and set flags Immediate
	CMPI	Compare Immediate
	CMNI	Compare negative Immediate
Arithmetic Extended	ADD	Add Extended Register
	ADDS	Add and set flags Extended
	SUB	Subtract Extended Register
	SUBS	Subtract and set flags Extended
	CMP	Compare Extended Register
	CMN	Compare negative Extended
Arithmetic with Carry	ADC	Add with carry
	ADCS	Add with carry and set flags
	SBC	Subtract with carry
	SBCS	Subtract with carry and set flags
	NGC	Negate with carry
	NGCS	Negate with carry and set flags
Logical Register	**AND**	Bitwise AND
	ANDS	Bitwise AND and set flags
	ORR	Bitwise inclusive OR
	EOR	Bitwise exclusive OR
	BIC	Bitwise bit clear
	BICS	Bitwise bit clear and set flags
	ORN	Bitwise inclusive OR NOT
	EON	Bitwise exclusive OR NOT
	MVN	Bitwise NOT
	TST	Test bits
Logical Immediate	**ANDI**	Bitwise AND Immediate
	ANDIS	Bitwise AND and set flags Immediate
	ORRI	Bitwise inclusive OR Immediate
	EORI	Bitwise exclusive OR Immediate
	TSTI	Test bits Immediate
Shift Register Shift Immed	**LSL**	Logical shift left Immediate
	LSR	Logical shift right Immediate
	ASR	Arithmetic shift right Immediate
	ROR	Rotate right Immediate
	LSRV	Logical shift right register
	LSLV	Logical shift left register
	ASRV	Arithmetic shift right register
	RORV	Rotate right register
Move Wide Immed iate	**MOVZ**	Move wide with zero
	MOVK	Move wide with keep
	MOVN	Move wide with NOT
	MOV	Move register
Bit Field Insert & Extract	BFM	Bitfield move
	SBFM	Signed bitfield move
	UBFM	Unsigned bitfield move (32-bit)
	BFI	Bitfield insert
	BFXIL	Bitfield extract and insert low
	SBFIZ	Signed bitfield insert in zero
	SBFX	Signed bitfield extract
	UBFIZ	Unsigned bitfield insert in zero
	UBFX	Unsigned bitfield extract
	EXTR	Extract register from pair
Sign Extend	*SXTB*	Sign-extend byte
	SXTH	Sign-extend halfword
	SXTW	Sign-extend word
	UXTB	Unsigned extend byte
	UXTH	Unsigned extend halfword
Bit Operation	CLS	Count leading sign bits
	CLZ	Count leading zero bits
	RBIT	Reverse bit order
	REV	Reverse bytes in register
	REV16	Reverse bytes in halfwords
	REV32	Reverses bytes in words

FIGURE 2.41 The list of assembly language instructions for the integer operations in the full ARMv8 instruction set. Bold means the instruction is also in LEGv8, italic means it is a pseudoinstruction, and bold italic means it is a pseudoinstruction that is also in LEGv8.

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
ADD a, b, c  // a gets b + c
```

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favours regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

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

- Compiled ARMv8 code:

```
ADD t0, g, h    // temp t0 = g + h
ADD t1, i, j    // temp t1 = i + j
~~ADD~~ f, t0, t1  // f = t0 - t1
```

Register Operands

- Arithmetic instructions use register operands
- ARMv8 has a 32 × 64-bit register file
  - Use for frequently accessed data
  - 64-bit data is called a "doubleword"
    - 31 x 64-bit general purpose registers X0 to X30
  - 32-bit data called a "word"
    - 31 x 32-bit general purpose sub-registers W0 to W30
- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations

ARMv8 Registers

- X0 – X7: procedure arguments/results
- X8: indirect result location register
- X9 – X15: temporaries
- X16 – X17 (IP0 – IP1): may be used by linker as a scratch register, other times as temporary register
- X18: platform register for platform independent code; otherwise a temporary register
- X19 – X27: saved
- X28 (SP): stack pointer
- X29 (FP): frame pointer
- X30 (LR): link register (return address)
- XZR (register 31): the constant value 0

Register Operand Example

- C code:

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

  - f, …, j in X19, X20, …, X23

- Compiled ARMv8 code:

```
ADD X9, X20, X21
ADD X10, X22, X23
SUB X19, X9, X10
```

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte

Memory Operand Example

- C code:

```
A[12] = h + A[8];
```

  - h in X21, base address of A in X22
- Compiled ARMv8 code:
  - Index 8 requires offset of 64

```
LDR        X9,[X22,#64]
ADD        X9,X21,X9
STR        X9,[X22,#96]
```

96/8 = 12

A[1] ⇒ [x22, #8]

A[2] ⇒ [x22, #16]

A

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Immediate Operands

- Constant data specified in an instruction

```
ADDI X22, X22, #4
```

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^{2} + 0\times2^{1} + 0\times2^{0}$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1

$$x + \overline{x} = 1111\ldots111_2 = -1$$

$$\overline{x} + 1 = -x$$

- Example: negate +2
  - $+2 = 0000\ 0000\ \ldots\ 0010_{two}$
  - $-2 = 1111\ 1111\ \ldots\ 1101_{two} + 1$
    $= 1111\ 1111\ \ldots\ 1110_{two}$

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

- In ARMv8 instruction set
  - `LDURSB`: sign-extend loaded byte
  - `LDURB`: zero-extend loaded byte

ARMv8 R-format Instructions

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

- Instruction fields
  - opcode: operation code
  - Rm: the second regis ter source operand
  - shamt: shift amount (00000 for now)
  - Rn: the first register source operand
  - Rd: the register destination

R-format Example

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

```
ADD X9,X20,X21
```

1112_{ten}	21_{ten}	0_{ten}	20_{ten}	9_{ten}
10001011000_{two}	10101_{two}	000000_{two}	10100_{two}	01001_{two}

$1000\ 1011\ 0001\ 0101\ 0000\ 0010\ 1000\ 1001_{two} =$

$8B150289_{16}$

ARMv8 D-format Instructions

opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

- Load/store instructions
  - Rn: base register
  - address: constant offset from contents of base register (+/- 32 doublewords)
  - Rt: destination (load) or source (store) register number
- *Design Principle 3:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

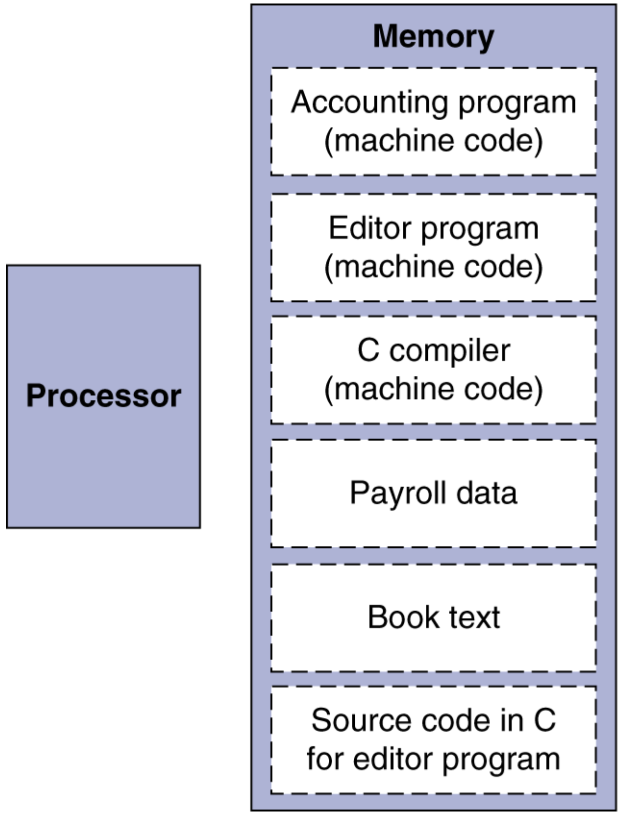
ARMv8 I-format Instructions

opcode	immediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits

- Immediate instructions
  - Rn: source register
  - Rd: destination register
- Immediate field is zero-extended

Stored Program Computers

The BIG Picture



Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program
Processor

- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

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)

  Any register may be used as base address

Reading Memory

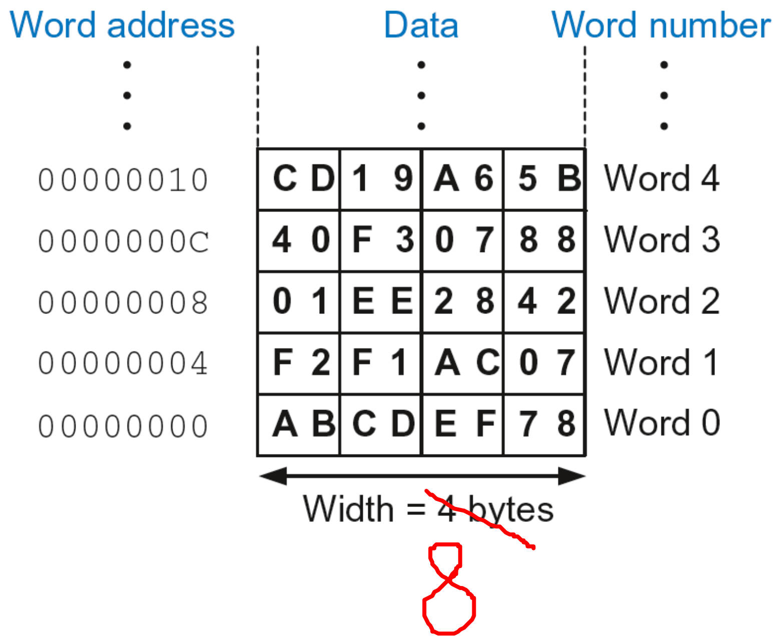
- **Example:** Read a word of data at memory address 8 into R3

Reading Memory

- **Example:** Read a word of data at memory address 8 into R3
  - Address = (X2 + 8) = 8
  - X3 = 0x01EE2842 after load

ARM Assembly Code

```
MOV X2, #0
LDR X3, [X2, #8]
```



Writing Memory

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

Writing Memory

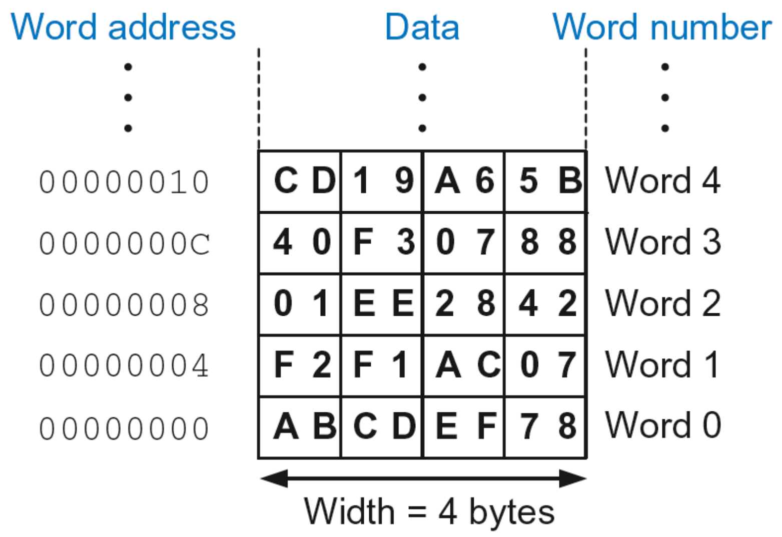
- **Example:** Store the value held in R7 into memory word 21.

Writing Memory: For 32 bit register set(ARMv7)

- **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]
```



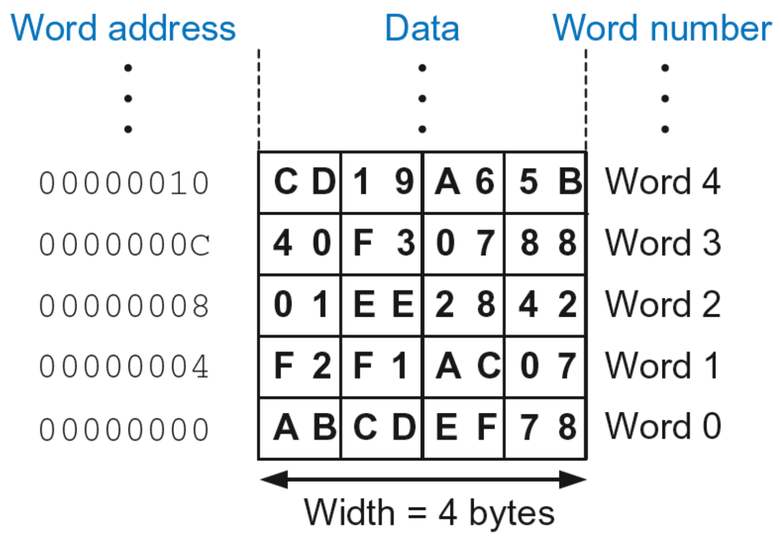
Writing Memory: For 32 bit register set(ARMv7)

- **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 in ARMv7 by 8 in ARMv8
- **Examples:**
  - Address of memory word 2 = 2 × 4 = 8 (ARMv7)
  - Address of memory word 10 = 10 × 4 = 40 (ARMv7)
  - Address of memory word 2 = 2 × 8 = 16 (ARMv8)
  - Address of memory word 10 = 10 × 8 = 80 (ARMv8)

Programming

High-level languages:

- e.g., C, Java, Python
- Written at higher level of abstraction

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` $\leftarrow$
- `ORR` $\leftarrow$
- `EOR` (XOR) $\leftarrow$
- `BIC` (Bit Clear)
- `MVN` (MoVe and NOT)

Logical Instructions: Examples ARMv7

Source registers

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

Assembly code		Result			
AND R3, R1, R2	R3	0100 0110	1010 0001	0000 0000	0000 0000
ORR R4, R1, R2	R4	1111 1111	1111 1111	1111 0001	1011 0111
EOR R5, R1, R2	R5	1011 1001	0101 1110	1111 0001	1011 0111
BIC R6, R1, R2	R6	0000 0000	0000 0000	1111 0001	1011 0111
MVN R7, R2	R7	0000 0000	0000 0000	1111 1111	1111 1111

Logical Instructions: Uses

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

Logical Instructions: Uses ARMv7

- `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 ARMv7

- `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

Logical Instructions: Uses ARMv7

- `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 ARMv7

- `LSL`: logical shift left
- `LSR`: logical shift right
- `ASR`: arithmetic shift right
- `ROR`: rotate right

Shift Instructions ARMv7

- `LSL`: logical shift left

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

- `LSR`: logical shift right

- `ASR`: arithmetic shift right

- `ROR`: rotate right

Shift Instructions ARMv7

- `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

- `ROR`: rotate right

Shift Instructions ARMv7

- `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

Shift Instructions ARMv7

- `LSL`: logical shift left

  Example: `LSL R0, R7, #5  ; R0=R7 << 5` $* 2^5$

- `LSR`: logical shift right

  Example: `LSR R3, R2, #31 ; R3=R2 >> 31` $/ 2^{31}$

- `ASR`: arithmetic shift right

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

- `ROR`: rotate right

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

Shift Instructions: Example 1 ARMv7

- **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

2^{17}

Shift Instructions: Example 2 ARMv7

- **Register** shift amount (uses low 16 bits of register)

Source registers

R8	0000 1000	0001 1100	0001 0110	1110 0111
R6	0000 0000	0000 0000	0000 0000	0001 0100

Assembly code		Result			
`LSL R4, R8, R6`	R4	0110 1110	0111 0000	0000 0000	0000 0000
`ROR R5, R8, R6`	R5	1100 0001	0110 1110	0111 0000	1000 0001

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	ARMv8
Shift left	<<	<<	`LSL`
Shift right	>>	>>>	`LSR`
Bit-by-bit AND	&	&	`AND, ANDI`
Bit-by-bit OR	|	|	`OR, ORI`
Bit-by-bit NOT	~	~	`EOR, EORI`

- Useful for extracting and inserting groups of bits in a word

Shift Operations

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `LSL` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `LSR` by *i* bits divides by 2^i (unsigned only)

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

```
AND X9,X10,X11
```

```
X10  00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
X11  00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
X9   00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000
```

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

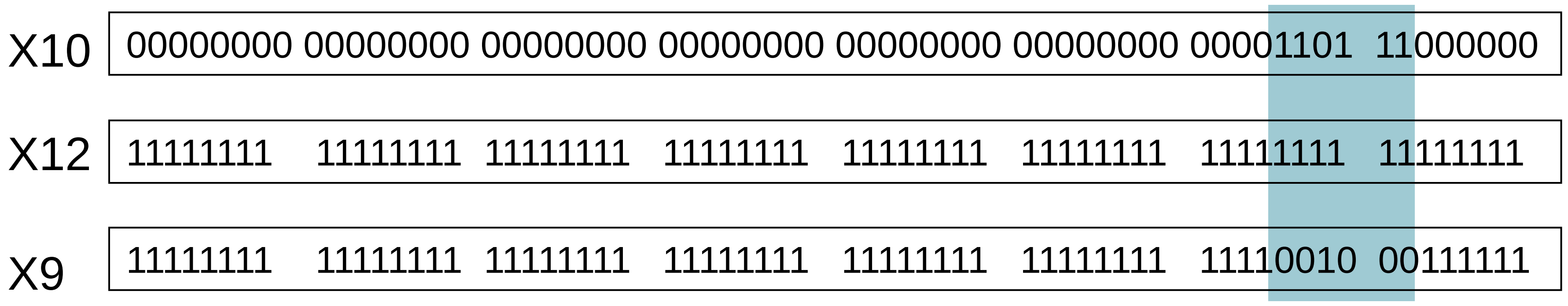
```
OR X9,X10,X11
```

X10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
X11	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
X9	00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000

EOR Operations

- Differencing operation
  - Set some bits to 1, leave others unchanged

```
EOR X9,X10,X12   // NOT operation
```



```
X10  00000000 00000000 00000000 00000000 00000000 00000000 00001101  11000000
X12  11111111   11111111  11111111   11111111   11111111   11111111   11111111   11111111
X9   11111111   11111111  11111111   11111111   11111111   11111111   11110010  00111111
```

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

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

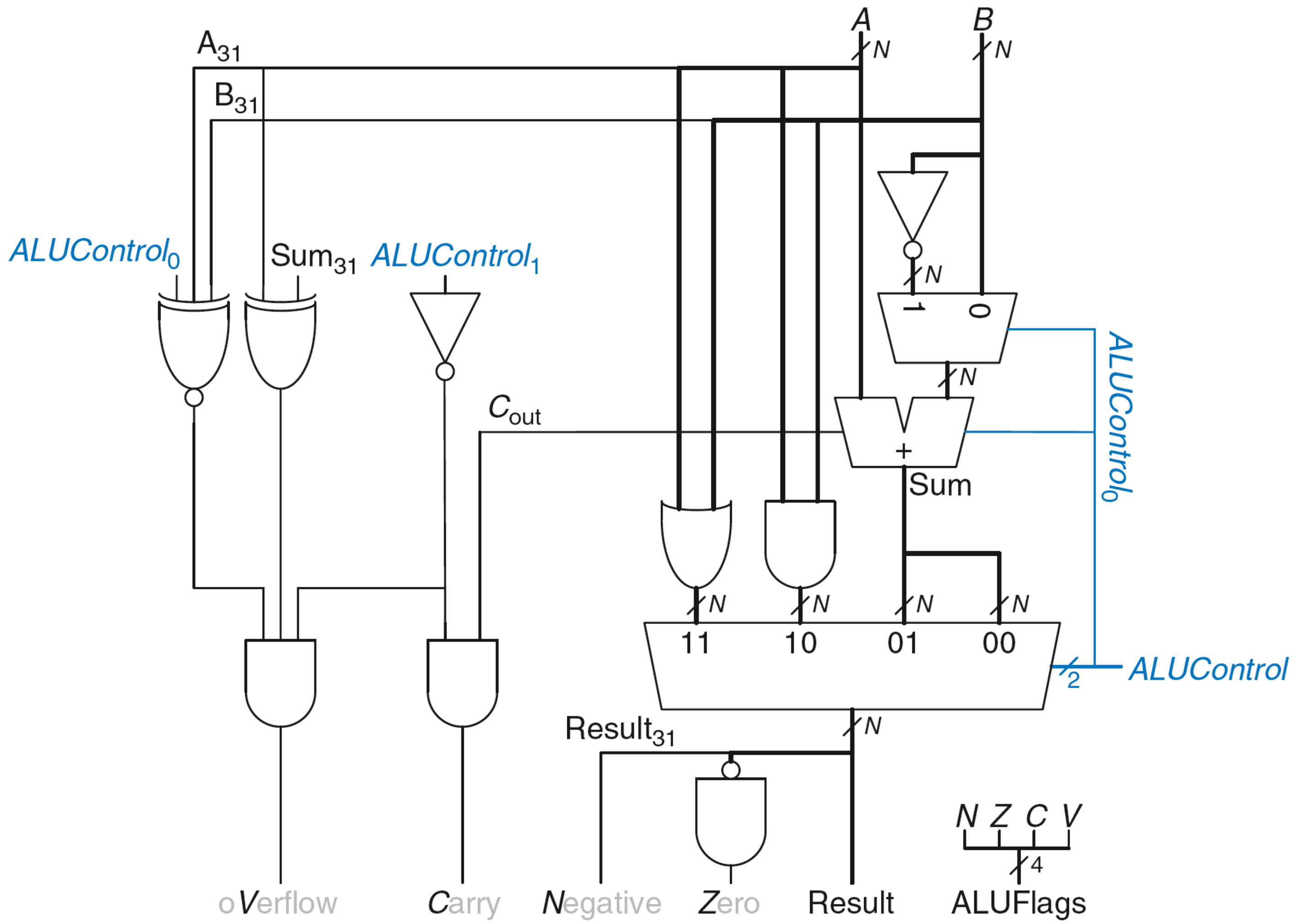
Flag	Name	Description
N	**N**egative	Instruction result is negative
Z	**Z**ero	Instruction results in zero
C	**C**arry	Instruction causes an unsigned carry out
V	o**V**erflow	Instruction causes an overflow

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

- Set by ALU
- Held in *Current Program Status Register* (*CPSR*)

Review: ARM ALU



Setting the Condition Flags: *NZCV*

- **Method 1:** Compare instruction: `CMP`

  Example: `CMP X5, X6`

  - Performs: X5-X6
  - Does not save result
  - Sets flags

Setting the Condition Flags: *NZCV*

- **Method 1:** Compare instruction: `CMP`

  Example: `CMP X5, X6`

  - Performs: X5-X6
  - 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*

- **Method 1:** Compare instruction: `CMP`

  Example: `CMP X5, X6`

  - Performs: X5-X6
  - Sets flags: If result is 0 (Z=1), negative (N=1), etc.
  - Does not save result
- **Method 2:** Append instruction mnemonic with **`S`**

Setting the Condition Flags: *NZCV*

- **Method 1:** Compare instruction: `CMP`
  - **Example:** `CMP X5, X6`
    - Performs: X5-X6
    - Sets flags: If result is 0 (Z=1), negative (N=1), etc.
    - Does not save result
- **Method 2:** Append instruction mnemonic with **`S`**
  - Example: `ADDS X1, X2, X3`
    - Performs: X2 + X3
    - Sets flags: If result is 0 (Z=1), negative (N=1), etc.
    - Saves result in ~~R~~1

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    X1, X2
  SUBNE  X3, X5, X8
  ```

  - **NE**: not equal condition mnemonic
  - `SUB` will only execute if X1 ≠ X2 (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

Conditional Execution

Example:

```
CMP    X5, X9              ; performs X5-X9
                           ; sets condition flags

SUBEQ X1, X2, X3           ; executes if X5==X9 (Z=1)
ORRMI X4, X0, X9           ; executes if X5-X9 is
                           ; negative (N=1)
```

if(x5==X9) {x1=X2-X3}
if (x5<x9) { ORR }

Conditional Execution

Example:

```
CMP   X5, X9           ; performs X5-X9
                       ; sets condition flags

SUBEQ X1, X2, X3       ; executes if X5==X9 (Z=1)
ORRMI X4, X0, X9       ; executes if X5-X9 is
                       ; negative (N=1)
```

Suppose X5 = 17, X9 = 23:

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

$\times$ SUBEQ **doesn't execute** (they aren't equal: *Z*=0)

$(X5 < X9) \rightarrow$ ORRMI **executes** because the result was negative (*N*=1)

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially

- `CBZ register, L1`
  - if (register == 0) branch to instruction labeled L1;

- `CBNZ register, L1`
  - if (register != 0) branch to instruction labeled L1;

- `B L1`
  - branch unconditionally to instruction labeled L1;

Compiling If Statements

- C code:

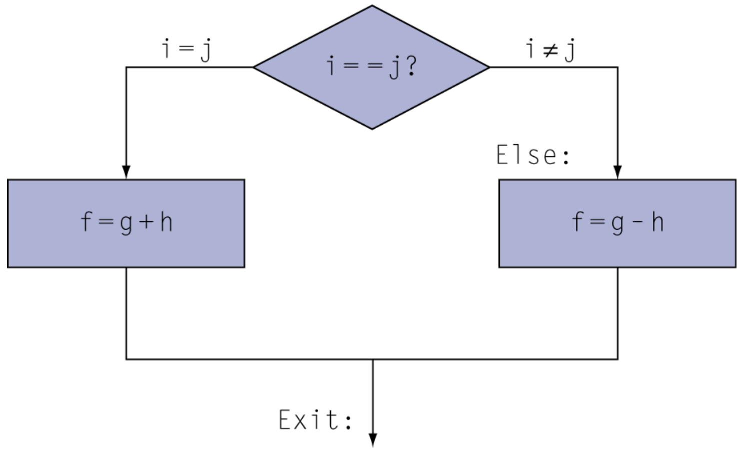
```
if (i==j) f = g+h;
else f = g-h;
```

  - f, g, … in X22, X23, …

- Compiled ARMv8 code:

```
         SUB X9,X22,X23
         CBNZ X9,Else
         ADD X19,X20,X21
         B Exit
Else:    SUB X9,X22,x23
Exit:  …
```

Assembler calculates addresses


i = j
i = = j?
i ≠ j
Else:
f = g + h
f = g - h
Exit: