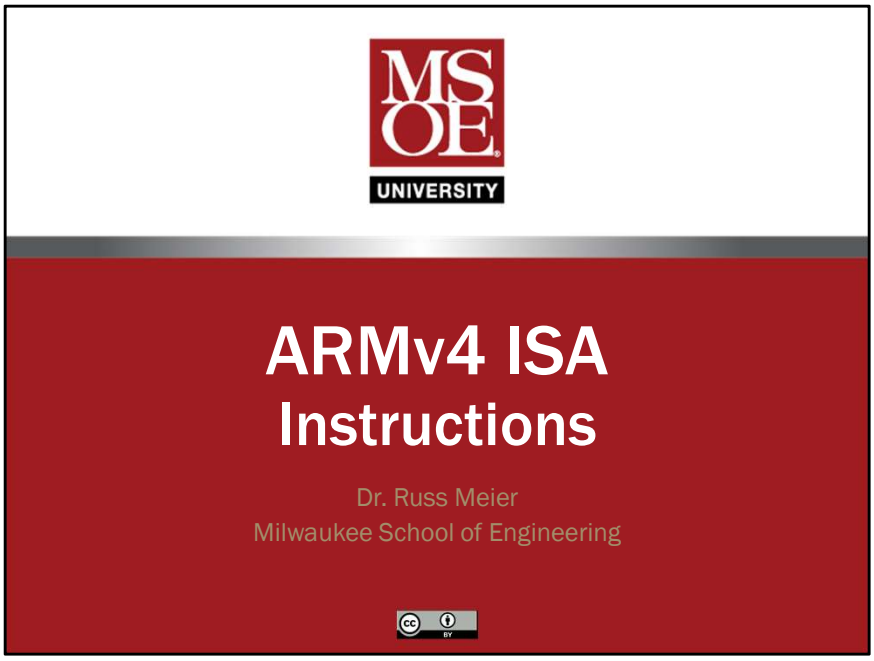
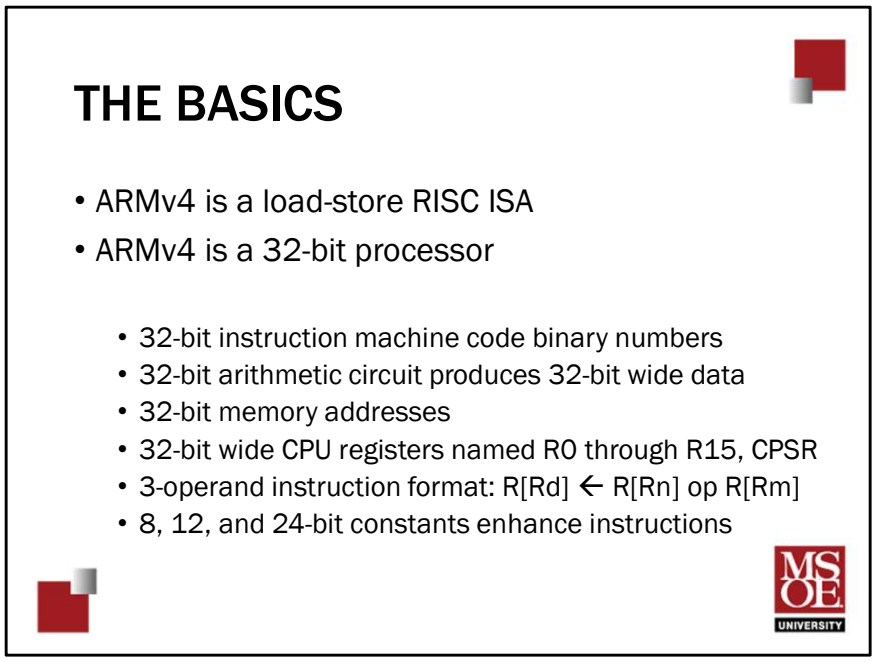

MSOE
UNIVERSITY
ARMv4 ISA
Instructions
Dr. Russ Meier
Milwaukee School of Engineering

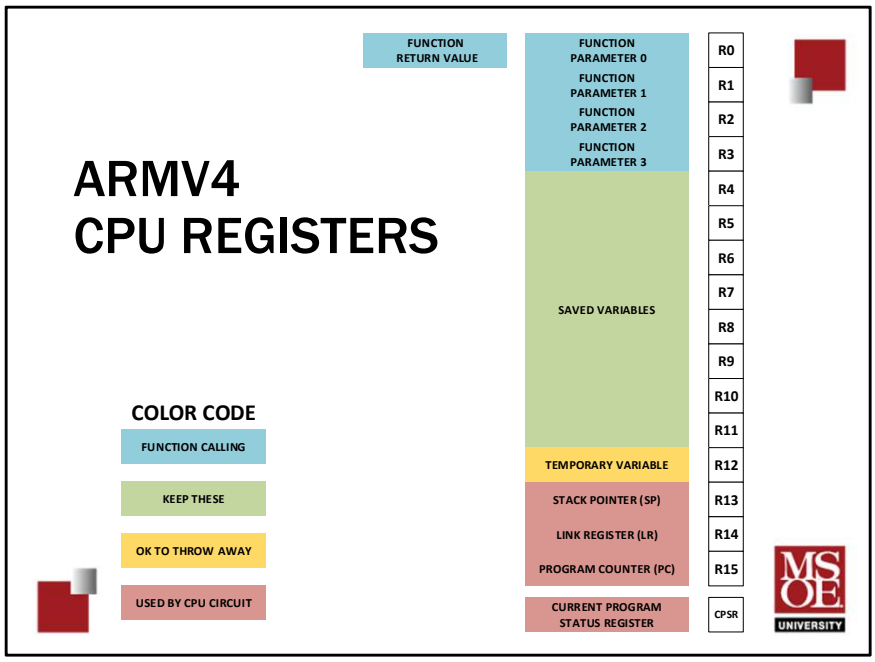



The ARM ISA was designed by Sophie Wilson at Acorn Computers, Limited – a British computer company – in 1985. Since then, the ARM ISA has evolved through multiple revisions. It is currently maintained and developed by ARM Holdings – a company with global headquarters in Cambridge, England. The version of ARM that we study in CE1921 is ARMv4 which was described in the mid 1990s.

- ARMv4 is a **load-store** RISC instruction set architecture. Load-store architectures do not allow arithmetic instructions to directly operate on memory. Instead, arithmetic instructions must be preceded by load instructions that move data into CPU registers. After these load instructions execute, arithmetic instructions can use the data. The arithmetic result is returned to a register and subsequent store instructions must be used to move data back down to main memory.
- ARMv4 is a **32-bit processor architecture**. This means that the data path moves 32-bit numbers through the arithmetic circuit. RISC architectures generally fix most numbers to the arithmetic width. ARMV4 is no exception – it has 32-bit fixed-width instruction binary numbers, 32-bit data, and 32-bit memory addresses.
- ARMv4 describes **seventeen CPU registers** to hold the 32-bit wide data brought into the processor from memory as well as the 32-bit wide results calculated by the circuits. Registers R0 through R12 are general-purpose registers available for

use by programmers. Registers R13, R14, R15, and the CPSR have special use in program sequencing.

- ARMv4 arithmetic instructions are **three-operand instruction format** with a destination register and two source locations specified.
- ARMv4 instructions use 8, 12, and 24-bit constants within instruction binary numbers to immediately provide constant values to the arithmetic circuits. This speeds computation by avoiding an extra load from data memory.

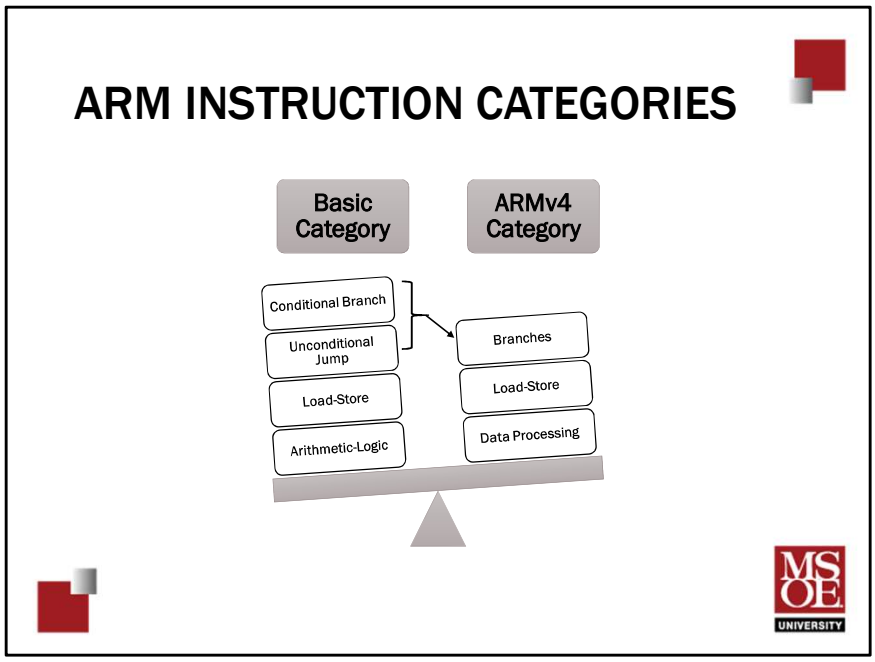



ARMV4 uses seventeen 32-bit wide registers to control program sequencing and hold user data.

- Registers R13, R14, R15 and the current program status register (CPSR) are part of the control circuitry used for program sequencing. These **special-purpose** registers are shaded in red because care must be taken to avoid incorrect changes to their stored values that could result in unexpected program behavior.
- Registers R0 through R12 are **general-purpose** registers used by the programmer to hold data.
- The ARMV4 ISA **recommends** the usage shown by the color bars. This recommendation is generally followed by compilers converting high-level language source code into assembly language. This recommendation is known as the **Procedure Call Standard**.

    - **Saved variables** are data values that the programmer intends to move down the memory pyramid for longer-term storage. Procedures – also called subroutines and functions – should not damage values that the main program may have stored in saved variable registers. Architects say that **saved variables are preserved across procedure calls**.

- **Temporary variables** are data values that are not important to the main program so any called procedure does not have to preserve the value of register R12. Architects say that temporary variables are **non-preserved registers**.
- **Function parameters** are values passed from a caller to a function.
- The **function return value** is the result of the function passed back to the caller.



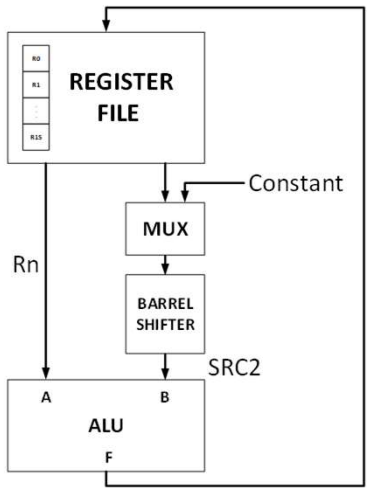

ARMv4 reduces the traditional four basic instruction categories into three.

- ARMv4 renames the arithmetic-logic category as **data processing.**
- ARMv4 combines conditional branches and unconditional jump instructions into a **single branch category** with both types of branches using the same instruction binary number format. This reduces complexity and introduces regularity to the instructions that change the program counter; an example of the modern design rule that **regular implies a simple and small design**.

## DATA PROCESSING INSTRUCTIONS

General Form: Rd ← Rn op Src2

- Register mode:
  Rd ← Rn op Rm
  ADD Rd, Rn, Rm
  ADD R0, R1, R2
  ADD R0, R1, R2, LSL #5
- Immediate mode:
  Rd ← Rn op constant
  ADD Rd, Rn, imm
  ADD R0, R1, #95
- Register-shifted-by-register mode:
  Rd ← Rn op (Rm shifted-by Rs)
  ADD Rd, Rn, Rm, shift-type Rs
  ADD R0, R1, R2, LSL R3




ARM data processing instructions are three-operand instructions specifying a destination register and two source data operands. The general form of all data processing instructions is shown. The **addressing mode** of the instruction describes what the operand called Src2 is.

- **Register mode** instructions have a second register as the second operand. This register can be shifted by some number of bits specified by a shift amount between zero and 31.
- **Immediate mode** instructions have a constant value as the second operand.
- **Register-shifted-by-register mode** instructions have a second register that is shifted by some amount specified in a fourth register. Shifting can be done to the left or to the right.

Color-coding has been used to help contextualize the levels of instruction abstraction.

- The most abstract view of the instruction is the **register-transfer level equation** shown in blue. This is also known as the **arithmetic architectural equation**.
- The next abstraction is an **instruction description** that uses a mnemonic for the desired operation and operand field names Rd, Rn, Rm, Rs, and immediate –

abbreviated as imm. These descriptions are shown in red.
- The least abstract version is the **actual instruction**. It is shown in green.

The arithmetic circuit diagram shows how the control circuit can use the addressing modes to control a multiplexer and a shifter in the data path for the second operand.

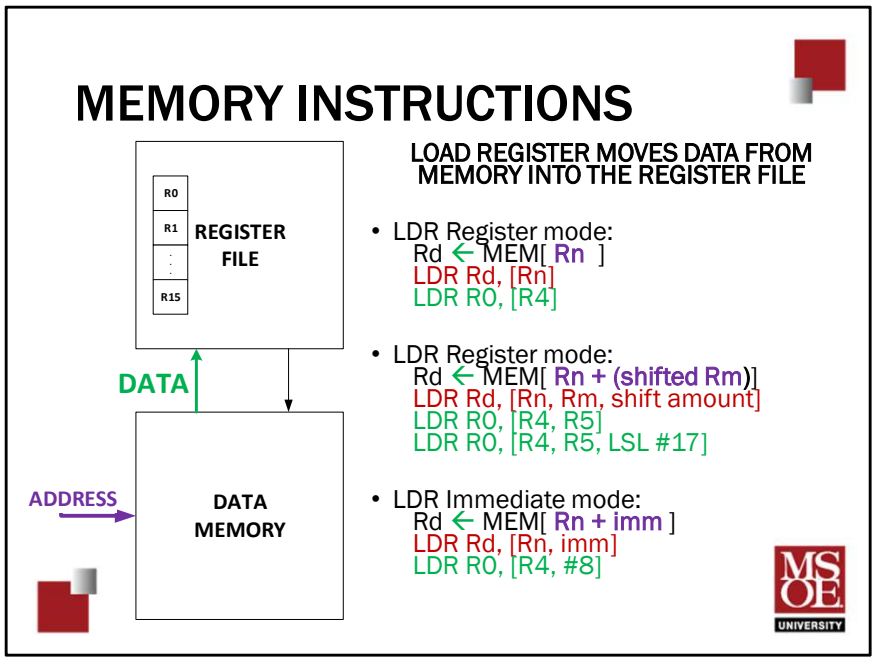
- **Immediate mode** instructions **will route the constant** through the multiplexer.
- **Register mode** instructions will **route the register** through the multiplexer.

## CORE DATA PROCESSING INSTRUCTIONS

| INSTRUCTION | MODE | EXAMPLE | REGISTER TRANSFER LEVEL BEHAVIOR (RTL) |
|---|---|---|---|
| ADD Rd, Rn, Rm | Register | ADD R3, R4, R5 | Rd ← Rn + Rm |
| ADD Rd, Rn, imm | Immediate | ADD R3, R4, #8 | R3 ← Rn + Extended-Immediate |
| AND Rd, Rn, Rm | Register | AND R8, R9, R10 | R8 ← Rn bitwise-and Rm |
| AND Rd, Rn, imm | immediate | AND R8, R9, #0xBE9 | R8 ← Rn bitwise-and Extended-Immediate |
| CMP Rn, Rm | register | CMP R4, R5 | Rn – Rm, capture ALU flags |
| CMP Rn, Imm | immediate | CMP R8, #0x91 | Rn – Extended-Immediate, capture ALU flags |
| EOR Rd, Rn, Rm | register | EOR R0, R10, R14 | Rd ← Rn bitwise-xor Rm |
| EOR Rd, Rn, imm | immediate | EOR R0, R10, #0x1A | Rd ← Rn bitwise-xor Extended-Immediate |
| MOV Rd, Rn, Rm | register | MOV R4, R5 | Rd ← Rm |
| MOV Rd, Rn, imm | immediate | MOV R4, #38 | Rd ← Extended-Immediate |
| ORR Rd, Rn, Rm | register | ORR R8, R9, R10 | Rd ← Rn bitwise-or Rm |
| ORR Rd, Rn, imm | immediate | ORR R1, R11, #15 | Rd ← Rn bitwise-or Extended-Immediate |
| SUB Rd, Rn, Rm | register | SUB R3, R4, R5 | Rd ← Rn – Rm |
| SUB Rd, Rn, imm | immediate | SUB R3, R4, #8 | Rd ← Rn – Rm |

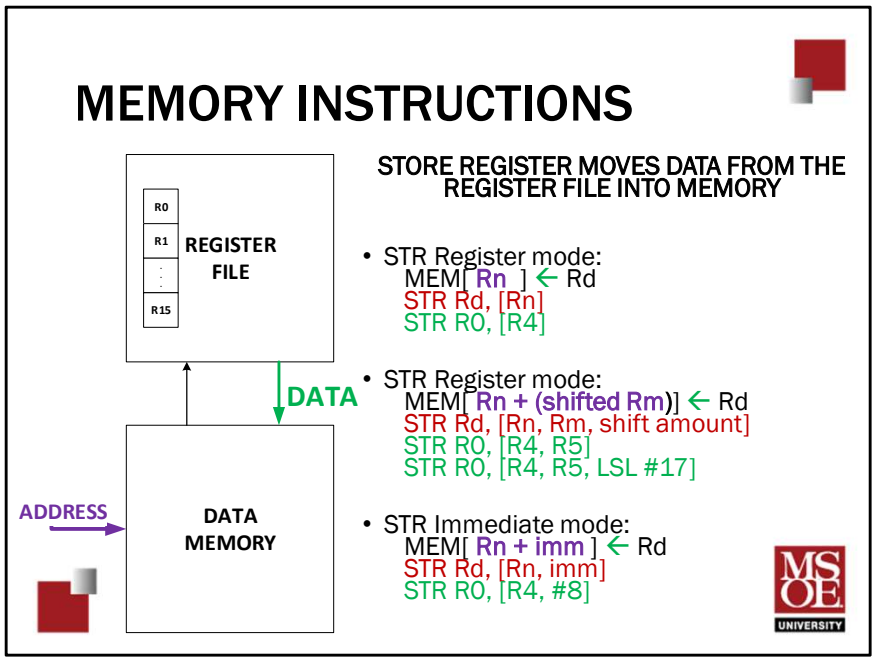
This table provides the **core set** of data processing instructions in register and immediate addressing modes. This small powerful set can be used to write general-purpose software.

- Note that this **core set** is **not the complete set** of ARM data processing instructions. The complete set can be found in Appendix B of the course textbook.
- Also note that register mode instructions can be modified to be register-shifted register mode. To keep the table shorter, register-shifted register mode instructions are not shown.

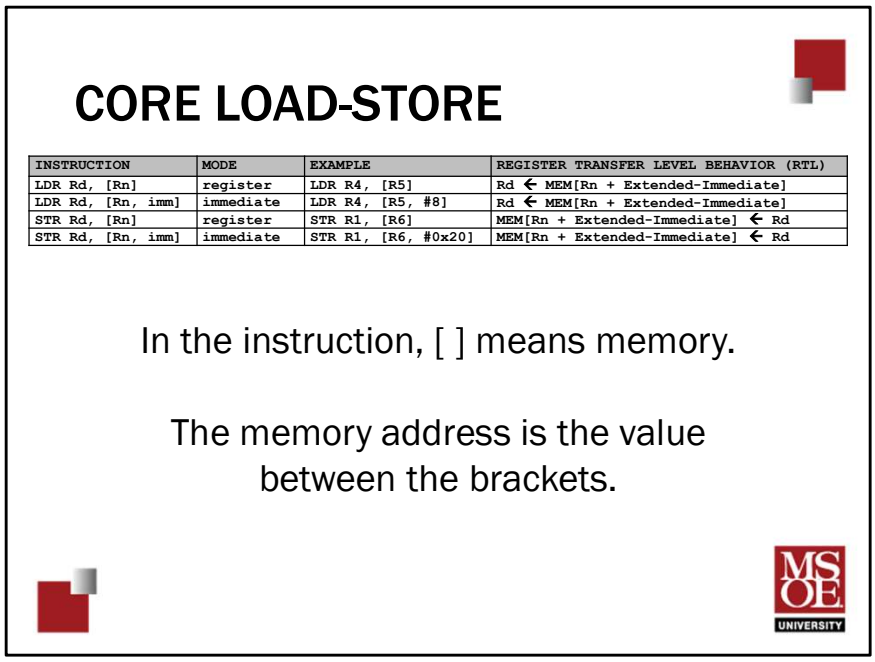



Memory instructions are **load-store** instructions that move data between the CPU and memory. The load-store instructions use register and immediate addressing modes to form operand 2. The load-store instructions do not support register-shifted-by register mode.

- The load instruction is called **load-register** and has the mnemonic **LDR**.
- LDR calculates a data memory address using the ALU, presents it to memory, and stores the 32-bit data provided by memory into the destination register.
- The memory address calculation is shown in purple in the register-transfer level equation. The arrows are also colored in green to help you visualize the direction of data flow.




- The store instruction is called **store-register** and has the mnemonic **STR**.
- STR calculates a data memory address using the ALU, presents it to memory, and routes the 32-bit data from the CPU register files to memory for storage.
- The memory address calculation is shown in purple in the register-transfer level equation. The arrows are also colored in green to help you visualize the direction of data flow.




This table summarizes the two 32-bit load-store instructions in their **basic** register and immediate addressing modes.

- ARM also includes instructions to move 8-bit values. The load-byte and store-byte instructions are called LDRB and STRB. Data flows in and out of the lower-byte of the specified CPU register.
- ARM also include instructions to move 16-bit values. The load-half-word and store-half-word instructions are called LDRH and STRH. Data flows in and out of the lower two bytes of the specified CPU register.

## ARMv4 CONDITIONAL EXECUTION

| SUFFIX | ALU CONDITION | CONDITION |
|---|---|---|
| EQ | A was equal to B | Z |
| NE | A was not equal to B | $\bar{Z}$ |
| CS | carry out was set to logic 1 | C |
| HS | unsigned number A was higher or the same as unsigned B | C |
| CC | carry out was cleared to logic 0 | $\bar{C}$ |
| LO | unsigned number A was lower than unsigned B | $\bar{C}$ |
| MI | result was negative (minus sign) | N |
| PL | result was positive or zero (plus sign) | $\bar{N}$ |
| VS | result overflowed and set the overflow bit to logic 1 | V |
| VC | no overflow and the overflow bit cleared to logic 0 | $\bar{V}$ |

```
CMP R10, R12
ADDEQ R0, R1, #95
```

Instruction set architectures use the arithmetic condition flags produced by the ALU to offer conditional execution of instructions.

- **Conditional execution** means an instruction only executes if the condition is true.
- ALUs announce arithmetic conditions on output bits called **flag bits**. The four standard ALU flag bits are carry out (**C**), signed overflow (**V**), negative number (**N**), and a zero result (**Z**). When considered as a nibble, CVNZ is called the **condition code nibble**, or simply the **condition code**.
- As seen on the data processing instruction summary slide, the **compare (CMP)** instruction captures the CVNZ bits and stores them in the current program status register (**CPSR**).

Many RISC ISAs use only a small set of condition suffixes to implement conditional branch instructions. The ARM ISA takes a very different approach. Almost all ARM instructions can have conditional suffixes appended to the mnemonic to control when the circuitry executes the instruction. The example on this slide compares the two numbers in R10 and R12 and only adds 95 to register R1 if R10 was equal to R12.

## ARMv4 CONDITIONAL EXECUTION

| SUFFIX | ALU CONDITION | CONDITION |
|---|---|---|
| HI | unsigned number A was higher than unsigned number B | $\bar{Z}C$ |
| LS | Unsigned number A was lower or the same as unsigned B | Z OR $\bar{C}$ |
| GE | signed number A was greater than or equal to signed B | $\overline{N \oplus V}$ |
| LT | signed number A was less than signed number B | N $\oplus$ V |
| GT | signed number A was greater than signed number B | $\bar{Z}(\overline{N \oplus V})$ |
| LE | signed number A was less than or equal to signed B | Z OR (N$\oplus$V) |
| AL | always execute the instruction / no condition | ignored |

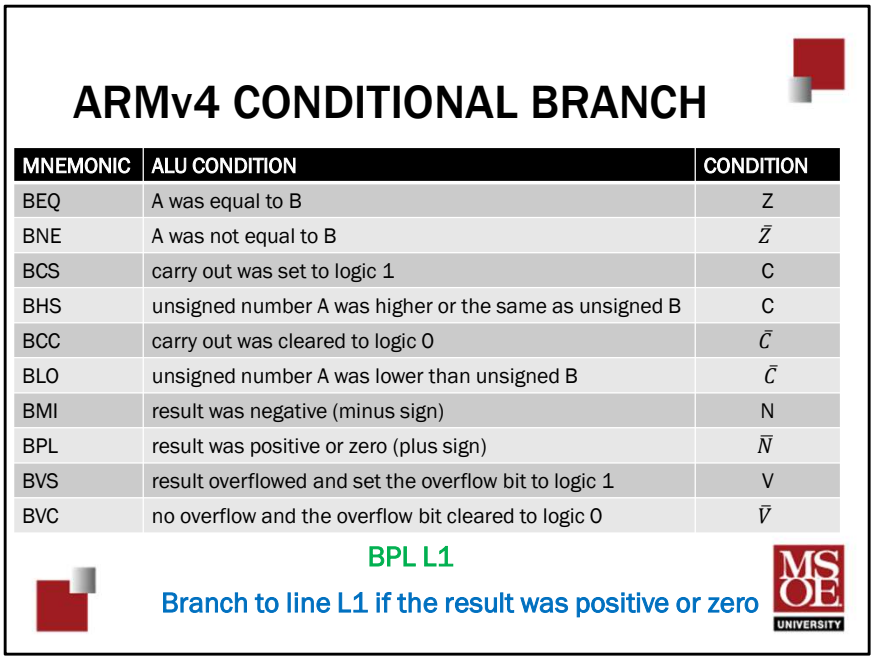
ADD R0, R1, R2 = ADDAL R0, R1, R2

The always suffix, AL, is normally omitted in ARM assembly language programs.

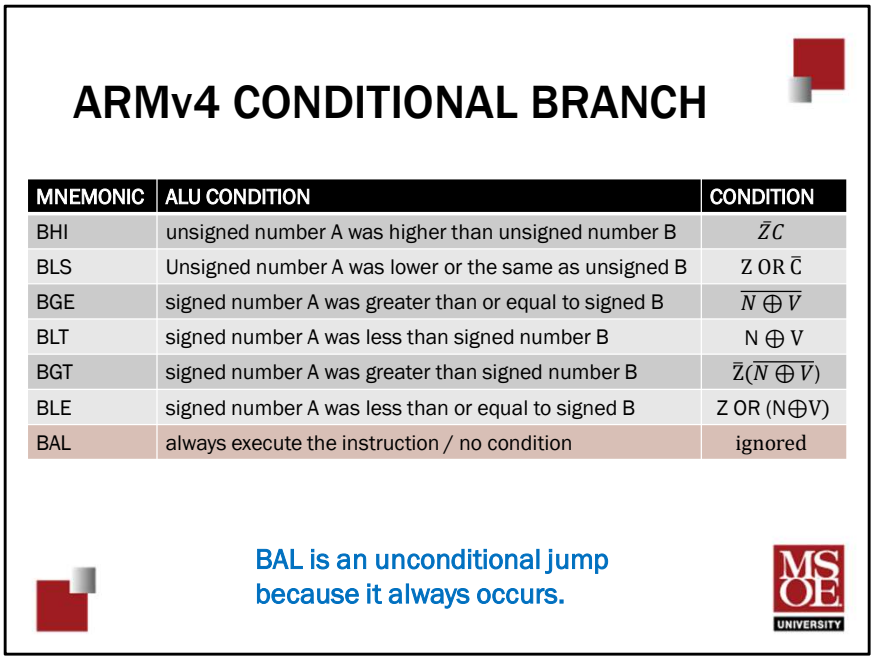
As you can see, ARM offers seventeen conditional mnemonic suffixes. Interestingly, some of the mathematical conditions use identical condition equations. There are fifteen unique condition equations. The ARMv4 ISA encodes the required condition equation – the mnemonic suffix if you will – as a **four-bit field** in the instruction binary number.

Instructions that are written without a conditional mnemonic suffix are unconditionally executed.

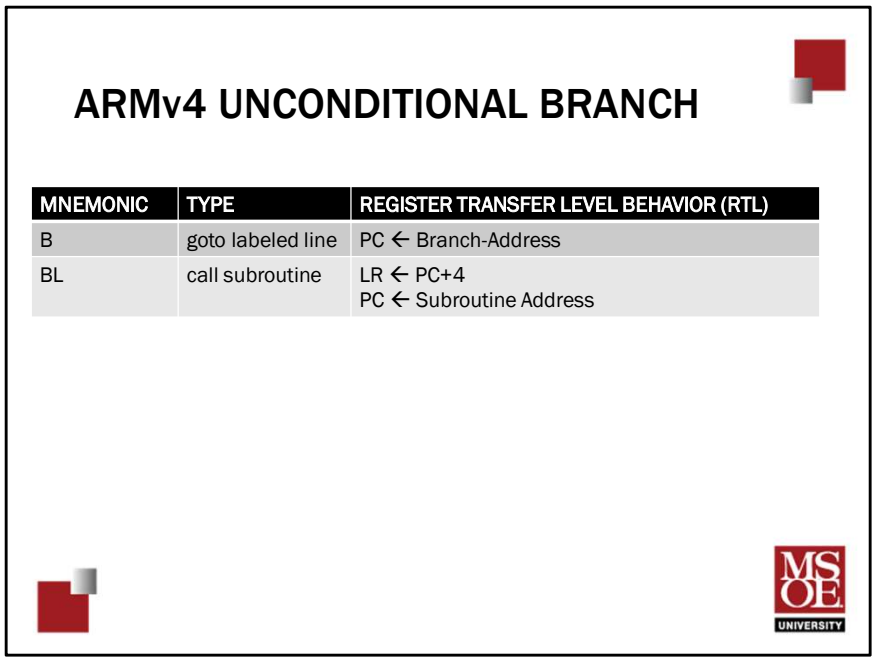
- In other words, these instructions **always execute**.
- ARMv4 does allow the always suffix, AL, to be added to mnemonics for emphasis but this is an exception to the normal practice of omitting the always suffix.



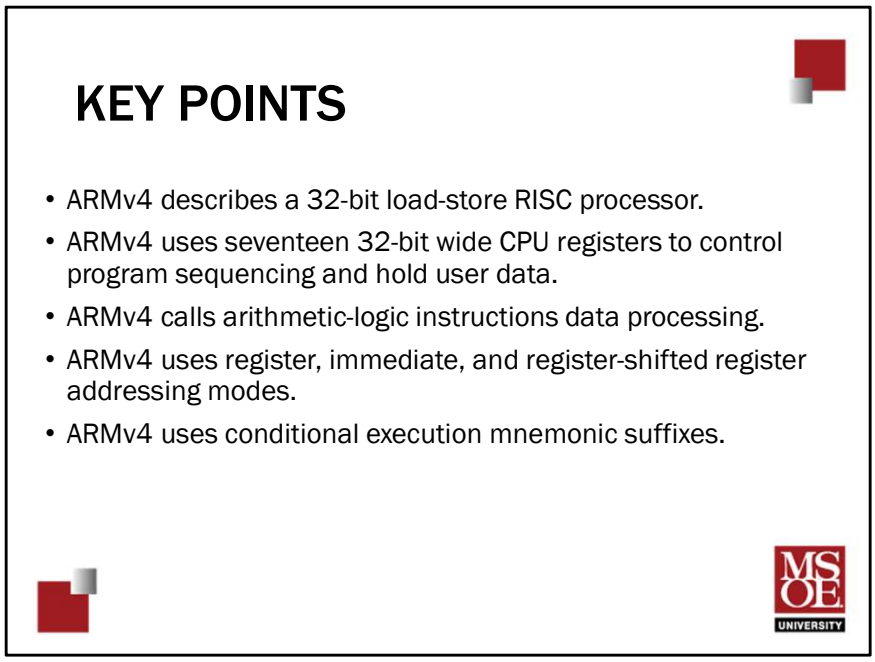

Assembly language programs and high-level language compilers implement for-loops, do-while loops, while-loops, and if-then-else algorithmic constructs using sequences of instructions and conditional branches. The green instruction shown on this slide is read as shown in blue.




Each of the condition mnemonic suffixes can be appended to the branch mnemonic. There are sixteen conditional branch instructions. The branch-always instruction is not a conditional instruction because it does not check the ALU condition code. It is highlighted in maroon on this slide to show the final conditional mnemonic suffix, but it would be categorized as an unconditional jump.




ARMv4 has two branch mnemonics, called **B and BL**, that can use the conditional mnemonic suffixes. When the always suffix, which is usually omitted, is used these mnemonics provide unconditional branches. The branch instruction, **B**, implements a goto that simply changes the program counter. The branch-and-link instruction, **BL**, implements a subroutine call by configuring the link register to allow the subroutine to return to the correct instruction when in finishes.




This summary slide notes the key points of this presentation. Continue to review this video when needed as your study the ARM instructions and use them to write assembly language programs.