CPE 221: Computer Organization

03 Instruction Set Architecture and ARM Basics

rahul.bhadani@uah.edu

THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Rahul Bhadani

Announcement

HW02: Due Jan 24, 2025, 11:59 pm.

THE UNIVERSITY OF
ALABAMA IN HUNTSVILLE

Architecture

Programmer's view of a computer

Instruction -> the language that computer understands

Instruction set -> vocabulary of the computer

Binary is used to encode instructions, same way words in English language is encoded in Alphabets.
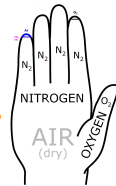
For us, humans to understand instructions, we code them using mnemonics called *Assembly language*.

THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Instruction Sets

We can have many instruction sets just like there are many languages.

E.g. ARM, x86, MIPS, SPARC, RISC V, etc.

Hardware ≠ Architecture

E.g. Intel and AMD both have x86 pc but different hardware specifications.

Rahul Bhadani

THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

arm

ARM designs computer architecture but doesn't build one.
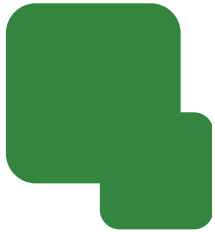
It sells licenses to other companies to build.

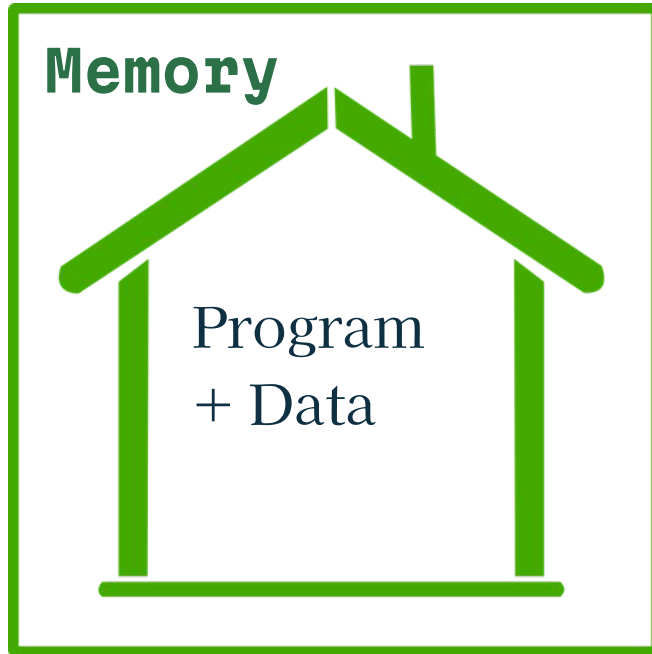**A**dvanced **R**ISC **M**achines Ltd, formerly known as Acorn RISC Machine

SAMSUNG
삼성전자

NVIDIA

HUAWEI

TEXAS INSTRUMENTS

ALTERA

Qualcomm

MEDIATEK

mi xiaomi

Spring 2025

Rahul Bhadani

THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Stored Program Machine

Stored Program Architecture

ARM:

Memory

Program
+ Data

Operates in fetch-execute mode where :
1. Instructions are read from the memory
2. Decoded
3. Executed

THE UNIVERSITY OF
ALABAMA IN HUNTSVILLE

Fundamental Structure of a Computer

Most basic Computer = Register + ALU + Buses

CPU

Register R1

Register R2

Register R4

Register R3
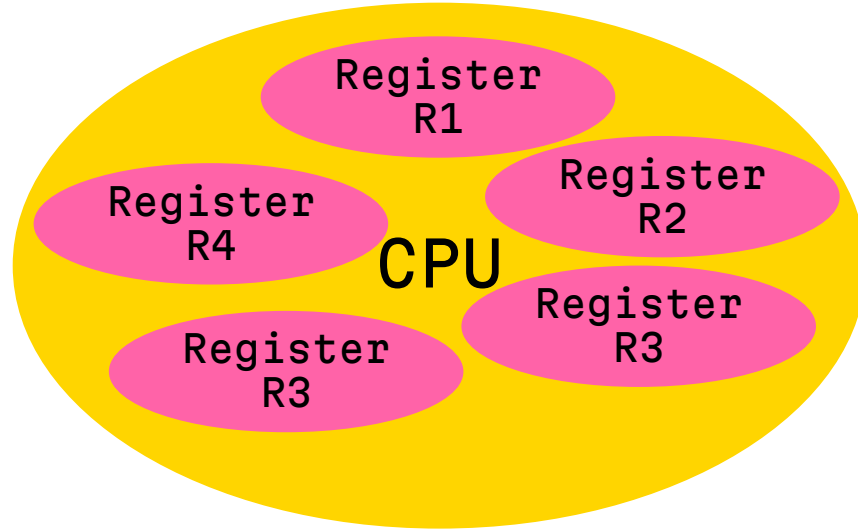
Register R3

15-32 registers
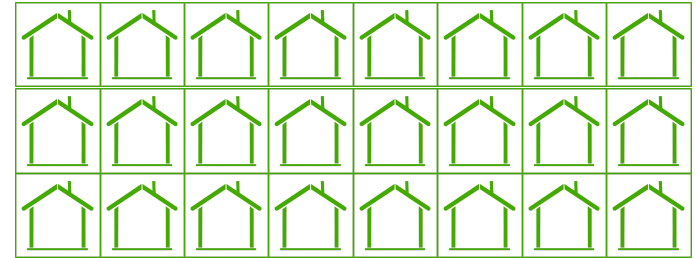
Memory much larger
Address : 32 bit or 64 bit

Assembly Language and Instructions

Human-readable version of the native language of computers.

Instructions are common operations that a computer is supposed to do.

ADD a, b, c $\longrightarrow$ $\longrightarrow$ 0x BD EF ...

Assembler

THE UNIVERSITY OF
ALABAMA IN HUNTSVILLE

Instructions in Memory and Special Registers

- Each instruction (like ADD 3 + 4) is stored as a binary string in memory.

- Which instruction is to be executed next is decided by a special register called **Program Counter (PC)**.

- Also known as **instruction pointer**.

PC ⟶

0x5622AFB0
0x5622AFB4
0x5622AFB8

THE UNIVERSITY OF
ALABAMA IN HUNTSVILLE
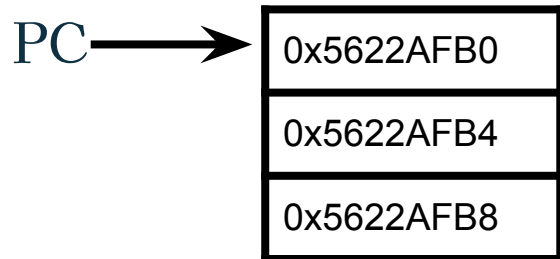
16 ARM Registers. R0 - R12 for storing variables.

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

THE UNIVERSITY OF
ALABAMA IN HUNTSVILLE

ARM Instructions

<instruction> <destination>, <source1>, <source2>

They are called immediate(s) or immediate operands.

They are preceded by a symbol *#*, e.g. *#10.*

Memory and Addresses

In ARM, instructions operate exclusively on registers, so data stored in memory must be moved to a register.

- ARM uses **32-bit memory addresses**, and **32-bit data words**.

- ARM ha a byte-addressable memory. Each byte in a memory has a unique address.

- 32-bit word = Four 8-bit bytes. Hence, each word address is a multiple of 4.

THE UNIVERSITY OF
ALABAMA IN HUNTSVILLE

ARM Architecture in Summary

- 16 registers called register set or register file.

- Arm instructions can also constants. They are called immediates or immediate operands. They don't require registers or memory access.

- Instructions operate exclusively on registers

- Data stored in the memory must be moved to register first before it can processed.

THE UNIVERSITY OF
ALABAMA IN HUNTSVILLE

Memory Map

Byte Address

⋮

Byte Address

⋮

13	12	11	10
F	E	D	C
B	A	9	8
7	6	5	4
3	2	1	0

0000 0010

0000 000C

0000 0008

0000 0004

0000 0000

MSB   LSB

THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Memory Map

Data

$\vdots$

Word Number

$\vdots$

CD	19	A6	5B	Word 4
40	F3	07	88	Word 3
01	EE	28	42	Word 2
F2	F1	AC	07	Word 1
AB	CD	EF	78	Word 0

⟷

Width = 4 bytes

THE UNIVERSITY OF ALABAMA IN HUNTSVILLE
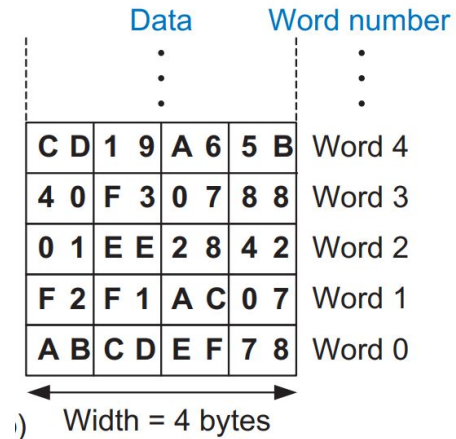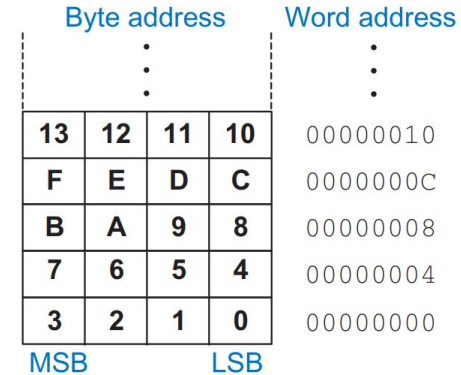
Memory in ARM Architecture

- ARM architecture has 32-bit memory addresses, 32-bit data word.

- Uses byte-addressable memory – each byte in the memory has a unique address

- 32 bit address = 8 bytes, each word address is a multiple of 4.

- MSB on the left side, LSB on the right side

- Low memory address at bottom, moving up towards high memory address.

Byte address				Word address
13	12	11	10	00000010
F	E	D	C	0000000C
B	A	9	8	00000008
7	6	5	4	00000004
3	2	1	0	00000000

MSB   LSB

Data				Word number
C D	1 9	A 6	5 B	Word 4
4 0	F 3	0 7	8 8	Word 3
0 1	E E	2 8	4 2	Word 2
F 2	F 1	A C	0 7	Word 1
A B	C D	E F	7 8	Word 0

Width = 4 bytes

Rahul Bhadani

Little Endian and Big Endian

Two ways byte-addressable memories are organized: Little-Endian, Big-Endian.

Example:

Suppose integer is stored as 4 bytes.

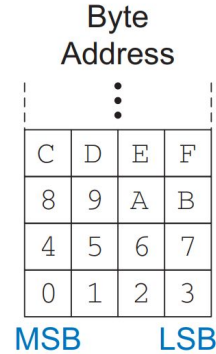If we have an integer variable x, whose value in hex is 0x01234567.

In Big Endian it is stored as

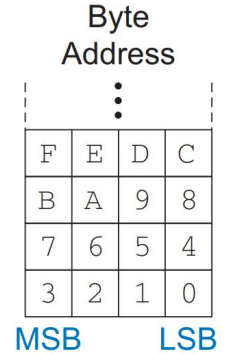|	0x100	0x101	0x102	0x103	
	01	23	45	67	

In Little Endian it is stored as

	0x100	0x101	0x102	0x103	
	67	45	23	01	

Big-Endian

Byte Address
⋮

C	D	E	F
8	9	A	B
4	5	6	7
0	1	2	3

MSB   LSB

Word Address
⋮

C
8
4
0

Little-Endian

Byte Address
⋮

F	E	D	C
B	A	9	8
7	6	5	4
3	2	1	0

MSB   LSB

Rahul Bhadani

THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Number of common instruction sets is kept small.

Less hardware required to decode the instruction.

E.g. 64 simple instructions only require 6 bits ($\log_2 64 = 6$)

Register Operations

ARMv7

ARMv7 has 16 general-purpose registers (R0-R15). Some of these have special purposes:

- R0-R12: General-purpose registers for data manipulation.

- R13 (SP): Stack Pointer (points to the top of the stack).

- R14 (LR): Link Register (stores the return address for functions).

- R15 (PC): Program Counter (points to the next instruction to execute).

Rahul Bhadani

Examples of Instructions

ADD

HLL (High-level Language) Code:

a = b + c

ARM Assembly Code:

ADD a, b, c

SUBTRACT

HLL (High-level Language) Code:

a = b - c

ARM Assembly Code:

SUB a, b, c   @ *a = b - c*

Start of the comment

Comment

THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

There are a few differences between book and how we will execute Assembly program in the CPUlator.

In the book,

OPCODE may be different.

Comments start with ;

In CPUlator, comments start with @

THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Register Specific Instructions

- MOV: Move data between registers or a constant into a register.

- LDR: Load data from memory into a register.

- STR: Store data from a register into memory.

Initialize using Immediates

High-level Code	ARM Assembly Code

`i  = 0;`
 	
`x = 4080;`
	`@ R4 = i, R5 = x`
	
	`MOV R4, #0 @ i = 0`
	
	`MOV R5, #0xFF0 @ x = 4080`
	
	The move instruction (MOV) is a useful way to initialize register values.

Example 1

ARM Assembly Code

```
.global _start
_start:

    MOV R0, #10



done:
    b done        @ after function return, infinitely loop
here
```

Entry point of the program

Example 2

```
MOV R1, R0    @copy the value from R0 to R1
```

LDR Syntax

ARM Assembly Code

```
LDR <destination register>, <memory address>
```

Copies the data from memory into register

Reading Memory

High-level Code	ARM Assembly Code

`a = mem[2];`	`@ R7 = a`
	`MOV R5, #0 @ base address = 0`
	`LDR R7, [R5, #8] @ R7 <= data at memory address (R5+8)`
	
	The LDR instruction specifies the memory address using a base register (R5) and an offset, which is 8 in this case. Each data word is 4 bytes, so word number 1 is at address 4, word number 2 is at address 8

Example 3: Load Operation

ARM Assembly Code

```
LDR R0, [R1] @load the value at memory address stored in R1
into R0
```

Rahul Bhadani

Example 4: Load with an Offset

```
LDR R0, [R1, #4]     @load the value at memory address (R1+4)
into 0
```

Example 5: Load from a Label or address

ARM Assembly Code

```
@ You can load data from a specific memory location defined by a label

.global _start
_start:

    LDR R0, =mydata @ load the address of mydata into R0
    LDR R1, [R0] @load the value at the address in R0 into R1

done:
    b done           @ after function return, infinitely loop here

.data
    mydata: .word 0x12345678 @Define a 32-bit word in memory
```

STR

```
@ Store Data to Memory

STR < source>, <memory address>
```

Writing to Memory

High-level Code	ARM Assembly Code

`mem[5] = 42;`	`MOV R1, #0 @ base address = 0`
	`MOV R9, #42`
	
	`@ value stored at memory address (R1+20) = 42`
	`STR R9, [R1, #0x14]`

ARM uses the store register instruction, STR, to write a data word from a register into memory.

Example 6: Store a value to a memory address

ARM Assembly Code

```
STR R0, [R1] @ store the value in R0 into the memory address
stored in R1
```
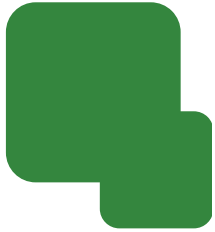
Example 7: Store with an offset

```
STR R0, [R1, #8] @ Store the value in R0 into the memory
address (R1+8)
```

Rahul Bhadani

Example 8: Store data to a label (address)
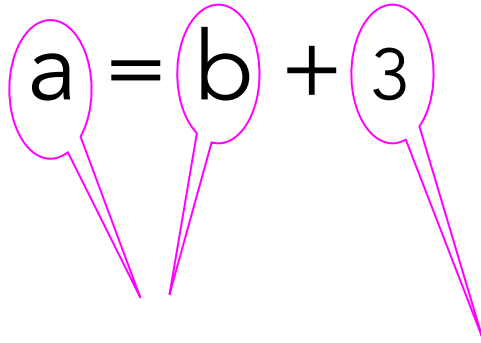
```
LDR R2, =mydata @load the address of `mydata` nto R2
STR R0, [R2]     @store the value in R0 into the memory
address stored in R2
```

Rahul Bhadani

Data Processing Instructions

Components of Instructions

$$a = b + 3$$

Operand

⬇

Stored in Register or Memory

Constant Operand

Stored in the instructions/part of the instruction

Accessing from registers is fast, but stores only small amount of data.

Accessing from memory is slow, can store a large amount of data.

Rahul Bhadani

THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Using Registers

High-level Code	ARM Assembly Code
`a = b + c;`	`@ R0 = a, R1 = b, R2 = c` `ADD R0, R1, R2 @ a = b + c`

Temporary Registers

High-level Code	ARM Assembly Code

`a = b + c - d;`	`@ R0 = a, R1 = b, R2 = c, R3 = d; R4 = t`
	`ADD R4, R1, R2 @ t = b + c`
	`SUB R0, R4, R3 @ a = t - d`

Rahul Bhadani

Immediate Operands (i.e. use of constants in instructions)

High-level Code
| ```
a = a + 4;
b = a - 12;
``` |

ARM Assembly Code
| ```
@ R7 = a, R8 = b
ADD R7, R7, #4 @ a = a + 4
SUB R8, R7, #0xC @ b = a - 12
``` |

Constants are written in Hex.

Rahul Bhadani

Logical Instructions

- AND, ORR (OR), EOR (XOR), and BIC (bit clear).

- These each operate bitwise on two sources and write the result to a destination register.

- The first source is always a register and the second source is either an immediate or another register.

- Another logical operation, MVN (MoVe and Not), performs a bitwise NOT on the second source (an immediate or register) and writes the result to the destination register.

- The bit clear (BIC) instruction is useful for masking bits (i.e., forcing unwanted bits to 0).

- The ORR instruction is useful for combining bitfields from two registers.

THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Bitwise Logical Operations

High-level Code	ARM Assembly Code
`r0 = r1 & r2;`	`AND r0, r1, r2`
`r0 = r1 | r2;`	`ORR r0, r1, r2`
`r0 = r1 ^ r2`	`EOR r0, r1, r2`
`r0 = r1 & ~r2`	`BIC r0, r1, r2 ;`

BIC stands for 'bit clear' – each '1' in r2 clears the corresponding bit in r1.

Rahul Bhadani

Shift Instructions

Shifts the values in a register left or right, dropping bits off the end.

A variant of the shift instruction, called rotate instruction doesn't drop bits off the end but rotate.

Left shift

High-level Code	ARM Assembly Code
`b = a << 2;`	`LSL R0, R5, #2   @logic shift left`

Logical Left shift.

R5 is shift by 2 places to the left and the value is stored in the register R0.

Rahul Bhadani

Right shift

High-level Code	ARM Assembly Code
`b = a >> 2;`	`LSR R0, R5, #2 @logic shift left` Logical Right shift. R5 is shift by 2 places to the right and the value is stored in the register R0.

Rotate

High-level Code

```
// Make sure the shift value is within [0, 31]
shift = shift % 32;
b = (a >> shift) | (a << (32 - shift));
```

C doesn't have equivalent rotate operator, but we can write a program to implement an equivalent operation. Alternative we can do in C++ using operator overloading.

ARM Assembly Code

```
ROR R3, R5, #21@ rotate right
```

There is no equivalent ROL, but left operation can be performed with right rotation by a complementary amount.

Arithmetic Shift Right vs Logical Shift Right

ARM Instruction sets also have ASR (Arithmetic Shift Right) which is slightly different from LSR.

In LSR, when bits are shifted, the new bits to the left is 0.

In ASR, when bits are shifted, the new bits to the left is filled with the same bit as original MSB.

Example. Let R5 = 1111000

LSR R2, R5 #2 will save 0011110 in R2

ASR R2, R5 #2 will save 1111110

THE UNIVERSITY OF
ALABAMA IN HUNTSVILLE

Multiply Instruction

Multiplying two 32 bit numbers may produce a 64 bit number.

We have two kind of multiply instructions in ARM

1. MUL – produces 32 bit results

2. UMULL – unsigned multiply long, produces 64 bit from unsigned numbers – stored in two registers

3. SMULL – signed multiply long, produces 64 bit from signed numbers – stored in two registers

THE UNIVERSITY OF
ALABAMA IN HUNTSVILLE

Multiply Instruction

High-level Code	ARM Assembly Code
`a = b*c;`	MUL R1, R2, R3 multiplies R2 and R3 and places the least significant 32 bit of the result in R1 and discard the most significant 32 bits

Rahul Bhadani

Multiply Instruction

High-level Code	ARM Assembly Code
`a = b*c;`	SMULL R1, R0, R2, R3 multiplies R2 and R3 and places the least significant 32 bit of the result in R1 and the most significant 32 bits is placed in R0. UMULL R1,R0, R2, R3

Multiply-accumulate Variant

High-level Code	ARM Assembly Code
`d = (a*c) + d`	`MLA R10, R1, R2, R3` to be multiplied to add The MLA instruction multiplies the values from R1 and R2, adds the value from R3, and places the least significant 32 bits of the result in R10.

SMLAL (Signed Multiply Accumulate Long)

High-level Code	ARM Assembly Code
We can have similar operation with UMLAL	SMLAL R1, R2, R3, R4 Most significant 32 bits Least significant 32 bits It basically does operation as [R2 R1] = [R2 R1] + (R3*R4)

Conditional Flags

Executing instruction, conditional depending on the value of flags.

Also known as status flags.
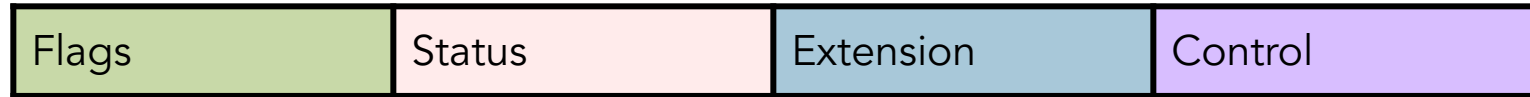
- Negative (N): Instruction result is negative, i.e. bit 31 of the result is 1.

- Zero (Z): Instruction result is zero.

- Carry (C): Instruction results in a carry out

- Overflow (V): Instruction causes an overflow

The most common way to set the status bits is with the compare (CMP) instruction, which subtracts the second source operand from the first and sets the condition flags based on the result. For example, if the numbers are equal, the result will be zero and the Z flag is set.

Rahul Bhadani

THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Current Program Status Register (CPSR)

The ARM core uses the CPSR to monitor and control internal operations. The CPSR is a dedicated 32-bit register and resides in the register file.

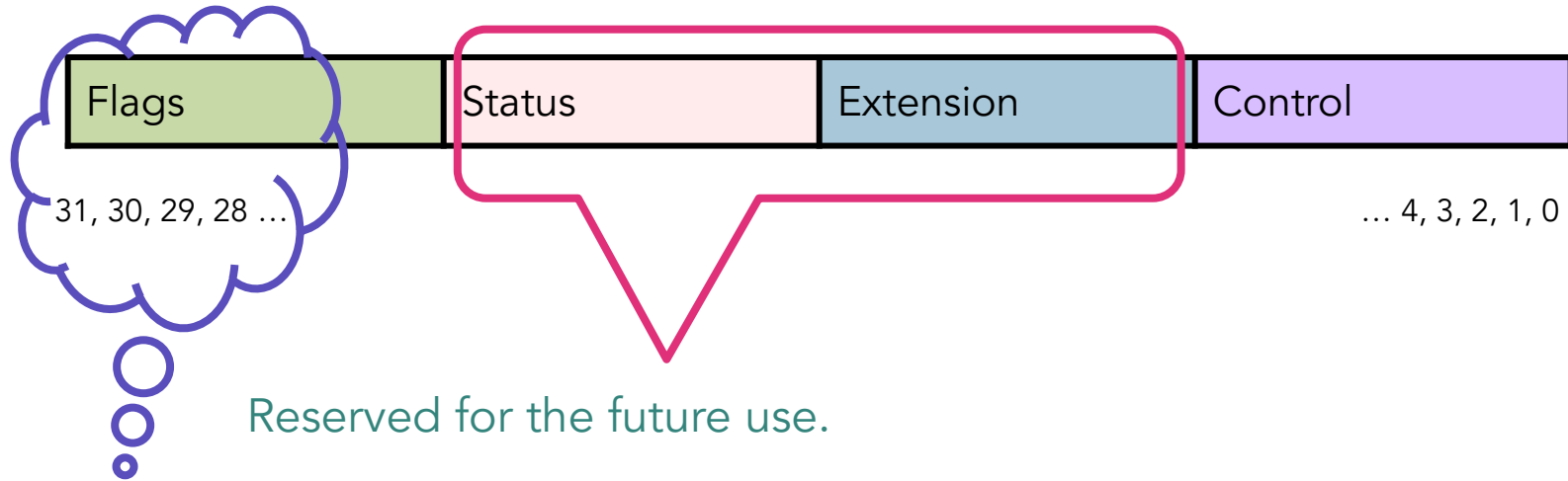The CPSR is divided into four fields, each 8 bits wide: flags, status, extension, and control.

Flags	Status	Extension	Control

31, 30, 29, 28 …   … 4, 3, 2, 1, 0

Rahul Bhadani

Current Program Status Register (CPSR)

Flags	Status	Extension	Control

31, 30, 29, 28 …   … 4, 3, 2, 1, 0
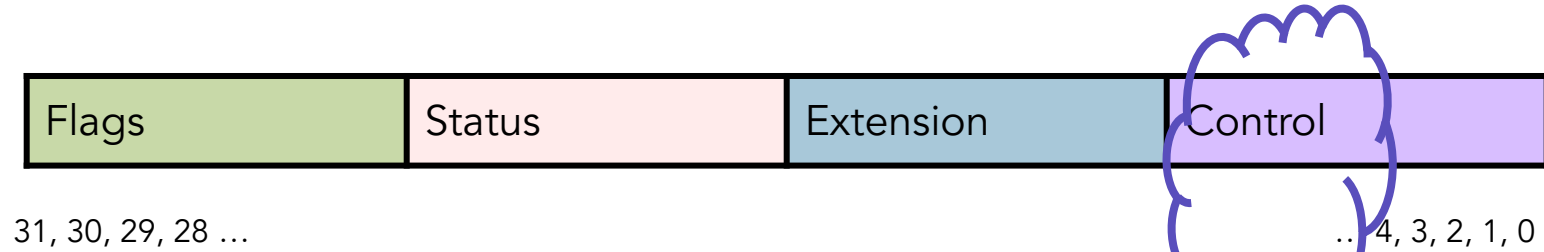
Reserved for the future use.

N Z C V Q IT[1:0] J

Together, they are called Application Program Status Register (APSR)

- N: Instruction result is negative, i.e. bit 31 of the result is 1.
- Z: Instruction result is zero. bit 30
- C: Instruction results in a carry out. bit 29
- V: Instruction causes an overflow. bit 28
- Q: Cumulative saturation/Sticky Flag. bit 27
- IT[1:0]: Reserved. bits 26, 25
- J: Reserved. bit 24. Some newer version indicates whether the core is in Jazelle state (we will discuss this later).

Rahul Bhadani

THE UNIVERSITY OF
ALABAMA IN HUNTSVILLE

Current Program Status Register (CPSR)

Flags	Status	Extension	Control

31, 30, 29, 28 …   .. 4, 3, 2, 1, 0

M = Processor Mode
T = indicates whether the core is in Thumb state.
F = disables Fast interrupt request (FIQ)
I = disables standard interrupt request (IRQ)
A = disables asynchronous aborts.

Includes the processor mode, status, and interrupt mask bits

A I F T M[4:0]

We will discuss this later

Example using Condition Flags

ARM Assembly Code

```
@ CMP (Compare)                        Subtracts the value of Operand2

@ Initialize R0, R1, and R2
MOV  R0, #10
MOV  R1, #10
MOV  R2, #20

@ Non-destructive subtract (R0, R1, R2 are not modified), update flags
registers
CMP  R0, #10          @ N=0, Z=1, C=1, V=0
CMP  R0, R1           @ N=0, Z=1, C=1, V=0
CMP  R0, R2           @ N=1, Z=0, C=0, V=0
```

Rahul Bhadani

Conditional Instructions

Using the status flags in the APSR, you can write assembly instructions that will conditionally execute. Conditional instructions allow us to implement high level language constructs like if/else and for loops.

We have Condition Mnemonics to implement conditional instructions in ARM ISA.

THE UNIVERSITY OF
ALABAMA IN HUNTSVILLE

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

Rahul Bhadani

THE UNIVERSITY OF
ALABAMA IN HUNTSVILLE

Data Processing Instruction with Condition Flags

Other data-processing instructions will set the condition flags when the instruction mnemonic is followed by "S." For example, SUBS R2, R3, R7 will subtract R7 from R3, put the result in R2, and set the condition flags.

All data-processing instructions will affect the N and Z flags based on whether the result is zero or has the most significant bit set. ADDS and SUBS also influence V and C, and shifts influence C.

Rahul Bhadani

THE UNIVERSITY OF
ALABAMA IN HUNTSVILLE

Type	Instructions	Condition Flags
Add	ADDS, ADCS	N, Z, C, V
Subtract	SUBS, SBCS, RSBS, RSCS	N, Z, C, V
Compare	CMP, CMN	N, Z, C, V
Shifts	ASRS, LSLS, LSRS, RORS, RRXS	N, Z, C
Logical	ANDS, ORRS, EORS, BICS	N, Z, C
Test	TEQ, TST	N, Z, C
Move	MOVS, MVNS	N, Z, C
Multiply	MULS, MLAS, SMLALS, SMULLS, UMLALS, UMULLS	N, Z

Rahul Bhadani

Example: Condition Execution

```
@R2 = 0x80000000

@R3 = 0x00000001

CMP R2, R3

ADDEQ R4, R5, #78

ANDHS R7, R8, R9

ORRMI R10, R11, R12

EORLT R12, R7, R10
```

Executes unconditionally, 0x80000000 – 0x00000001 = 0x80000000 + 0xFFFFFFFF = 0x7FFFFFFF, with C= 1, V = 1

ADDEQ and ORRMI do not execute because the result of R2 – R3 is not zero (i.e., R2 ≠ R3) or negative. ADDEQ onnly adds if Z =0. ORRRMI only executes of the result is negative.

Executes because R2 ≥ R3 (unsigned).

Executes because R2 < R3 (signed)

What does this program do, write a C-equivalent? (In-class Exercise)

ARM Assembly Code

```
MOV R0, #10


CMP R0, #5
MOVGT R0, #7
MOVLE R0, #0
```

Rahul Bhadani

What does this program do? (In-class Exercise Solution)

High-level Code	ARM Assembly Code

```
i = 10;

if(i>5)
   i = 7;
else
   i = 0;
```

```
@ Initialize R0
  MOV R0, #10

@ If Else
  CMP R0, #5
  MOVGT R0, #7
  MOVLE R0, #0
```

Branching

Most computer architectures including ARM architecture use branch instructions to skip over sections of code or repeat code.

A program usually executes in sequence, with the program counter (PC) incrementing by 4 after each instruction to point to the next instruction. (Recall that instructions are 4 bytes long and ARM is a byte-addressed architecture.) Branch instructions change the program counter.

Branch Instructions

B: Simple Branching

BL: Branch Link (used for function calls)

Branches can be unconditional or conditional. They are also called jumps in some other architectures.

Branch Instruction Example 1

```
ADD R1, R2, #17 @ R1 = R2 + 17
B TARGET @ branch to TARGET
ORR R1, R1, R3 @ not executed
AND R3, R1, #0xFF @ not executed
TARGET

SUB R1, R1, #78 @ R1 = R1 - 78
```

Kind of like GOTO in C

Branch label: cannot be a reserved keyword/mnemonics

Always executed, either after skipping ORR and AND, or after executing them

Rahul Bhadani

Branch Instruction Example 2

```
MOV R0, #4 @ R0 = 4
ADD R1, R0, R0 @ R1 = R0 + R0 = 8
CMP R0, R1 @ set flags based on R0-R1 = -4. NZCV = 1000
BEQ THERE @ branch not taken (Z != 1), branch dependent
on equality
ORR R1, R1, #1 @ R1 = R1 OR 1 = 9
THERE
ADD R1, R1, #78 @ R1 = R1 + 78 =
```

When code reaches the BEQ instruction, the Z = 0, i.e., R0 ≠ R1. So the branching doesn't happen. Hence ORR is executed. Execution never go to THERE

Rahul Bhadani

Conditional Statements

If/else,

switch/case

Example: if

High-level Code	ARM Assembly Code

```
if (apples == oranges)
      f = i + 1;



f = f - i;
```

```
@ R0 = apples, R1 = oranges, R2 = f, R3 = i
CMP R0, R1 @ apples == oranges ?
BNE L1 @ if not equal, skip if block
ADD R2, R3, #1 @ if block: f = i + 1
L1
    SUB R2, R2, R3 @ f = f - i
```

Example: if/else

High-level Code

```
if (apples == oranges)
    f = i + 1;
else
    f = f - i;
```

ARM Assembly Code

```
@ R0 = apples, R1 = oranges, R2 =
f, R3 = i
CMP R0, R1 @ apples ==
oranges?
BNE L1 @ if not equal, skip
if block
ADD R2, R3, #1 @ if block: f
= i + 1
B L2 @ skip else block
L1
SUB R2, R2, R3 @ else block:
```

Example: switch/case

High-level Code	ARM Assembly Code

```
switch (button) {
case 1: amt = 20; break;
case 2: amt = 50; break;
case 3: amt = 100; break;
default: amt = 0;
}
// equivalent function using
// if/else statements
if (button == 1)amt = 20;
else if (button == 2)amt = 50;
else if (button == 3) amt = 100;
else amt = 0;
```

```
@ R0 = button, R1 = amt
    CMP R0, #1 @ is button 1 ?
    MOVEQ R1, #20 @ amt = 20 if
    button is 1
    BEQ DONE @ break
    CMP R0, #2 @ is button 2 ?
    MOVEQ R1, #50 @ amt = 50 if
    button is 2
    BEQ DONE @ break
    CMP R0, #3 @ is button 3?
    MOVEQ R1, #100 @ amt = 100 if
    button is 3
    BEQ DONE @ break
    MOV R1, #0 @ default amt = 0
DONE
```

Repeatedly executing a certain task.

- while loop

- for loop

Loop Example: while loop

High-level Code	ARM Assembly Code

High-level Code:

```
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 @ pow != 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
```

Rahul Bhadani

Loop Example: for loop

High-level Code

```
int i;
int sum = 0;
for (i = 0; i < 10; i = i + 1) {
    sum = sum + i;
}
```

ARM Assembly Code

```
@ R0 = i, R1 = sum
    MOV R1, #0 @ sum = 0
    MOV R0, #0 @ i = 0 loop initialization
FOR
    CMP R0, #10 @ i < 10 ? check condition
    BGE DONE @ if (i >= 10) exit loop
    ADD R1, R1, R0 @ sum = sum + i loop body
    ADD R0, R0, #1 @ i = i + 1 loop operation
    B FOR @ repeat loop
DONE
```

Rahul Bhadani

Video of the day

How Amateurs created the world´s most popular Processor (History of ARM Part 1)

https://www.youtube.com/watch?v=nIwdhPOVOUk