Instruction Set Architecture

Each processor has a predefined set of instructions that it implements called the *instruction set*.

Instruction Set Architecture (ISA)

The ISA, or programming model, consists of the instruction set, information about how memory is organized, how to access memory, etc.

Instruction Set Architecture

- The ISA forms a contract between the machine and the programmer, defining features that
  - software *may* use, and
  - hardware *will* implement
- In general, multiple processors implement any given ISA
  - *E.g.*, consider: x86-64, ARMv7-A, Power ISA 3.0, RISC-V

Note: the ISA need not define how hardware will implement any given feature.

Different Implementations of an ISA

Machine language software (assembly) is portable between two processors if they implement the same ISA.

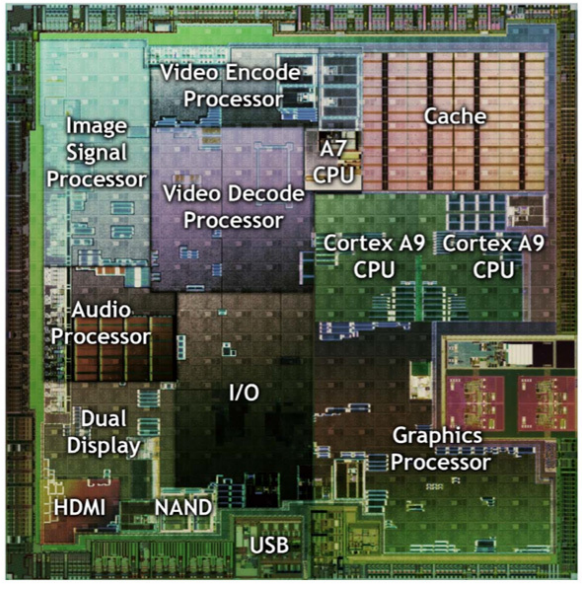
- The ISA is the interface between the hardware and software
- The ISA tells you what the processor does; the ISA is a public specification
- The implementation is how it does it; the implementation is private (trade secrets, etc)

ISAs may be used for a long time because of legacy software.

- x86 was introduced in 1978
- x86-64 extended x86 to support 64-bit operations in 2001
- Consequently, x86 software written for the 8086 in 1978 runs on Core i7 (x86-64) in 2021

The ARM Architecture

- A family of RISC processors used in many devices, especially smartphones and tablets
- There have been 200 billion ARM processors shipped as of 2021[a] (∼29 billion for 2021 alone[b])
- ARM provides the processor design to chip manufacturers, who fabricate it in their own products:
  - *e.g.*, Apple A5 chip has a dual-core ARM Cortex-A9 processor
  - *e.g.*, Nvidia Tegra 2 SoC also has the same ARM processor



Nvidia Tegra 2 SoC

source: www.anadtech.com

[a] https://www.arm.com/blogs/blueprint/200bn-arm-chips

[b] https://www.statista.com/statistics/1131983/arm-based-chip-unit-shipments-as-reported-by-licensees-worldwide/

ARM ISA Basics

The word length is 32 bits; processor registers are 32 bits; the address size is 32 bits.

The ISA is (mostly) RISC:

- All* instructions are 32-bits long.
- Only load and store instructions access memory.
- All arithmetic and logic instructions operate on registers.
- There are some features which normally are seen in CISC ISAs.

* *The ARM ISA also supports 16-bit wide Thumb-2 instructions.*

ARM ISA Memory

- Memory is byte-addressable using 32-bit addresses
- Memory is *litte-endian*
- Word, half-word, and byte data transfers to and from processor registers are supported (SW's perspective)
- All memory accesses are word-aligned (HW's implementation)

ARM Programmer-visible Registers

ARM implements sixteen 32-bit processor registers labeled R0 through R15.

- R15 is the program counter (PC)
- R14 is the link register (LR)
- R13 is the stack pointer (SP)

In general, we use only* R0...R12 as General Purpose Registers (GPRs) and only use and refer to R13, R14, and R15 as **SP**, **LR**, and **PC**.

* *In practice, additional guidelines further limit the use of registers by programmers and compilers. Curious? See the ARM Architecture Procedure Call Standard.*

There is also a special status register called the Current Program Status Register (CPSR) that indicates various useful information (more later).

Assembly Language Syntax

Assembly language consists of shorthand instruction names called mnemonics, a syntax for using them, and other directives for organizing them.

A program called an assembler translates the mnemonics into machine language instructions (binary; more later).

Here is a (short) ARM assembly program:

```
ADD  R1, R2, R3   // R1 <-- R2 + R3
```

- `ADD` is a mnemonic
- `R1` is a destination register; the first operand
- `R2` and `R3` are source registers; the second and third operand
- `// R1 <-- R2 + R3` is a comment (not a very useful one)

There are different ways to use each instruction.

```
ADD  R1, R2, R3   // R1 <-- R2 + R3
```

Here, the syntax of the instruction is `ADD Rd, Rn, Rm` where

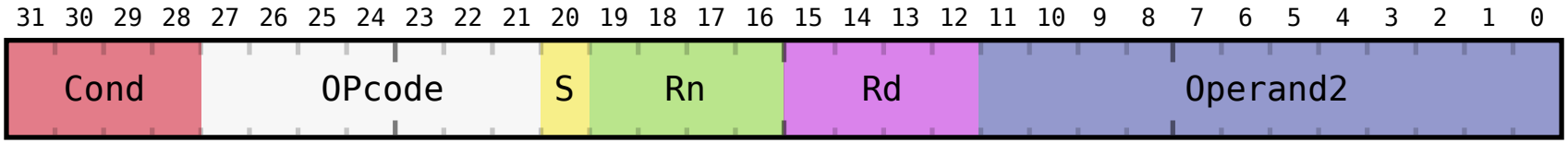
- `Rd` specifies the destination register
- `Rn` and `Rm` specify the source registers

```
ADD  R4, R5, #24   // R4 <-- R5 + 24
```

Here, the syntax of the instruction is `ADD Rd, Rn, Imm` where

- `Rd` specifies the destination register
- `Rn` specifies the source register
- `Imm` specifies an immediate value (constant)

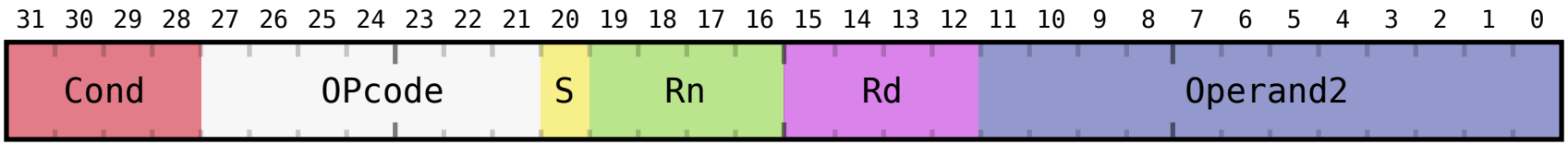
Instruction Format and Operands



Assembly instructions ultimately become machine instructions; above, a 32-bit instruction is divided into several fields that determine its operation:

- Cond: condition codes; more later
- OPcode: specifies the operation to be executed
- Rn, Rd, Operand2: operands the operation works with/on

Instruction Format and Operands



Each operand has a limited set of allowable uses:

- `Rd` refers to a destination register to which results are written
- `Rn` and `Rm` refer to source registers; their value does not change (unless the register is the same as `Rd`)
- `Imm` refers to an immediate value (the maximum number of bits might be specified, *e.g.*, `Imm16` for a 16-bit value); immediates are saved in the instruction itself
- `Op2` refers to a flexible source operand, which is either:
  - an 8-bit immediate value `Imm8`
  - a register (with optional rotation or shift)

Move Instructions

These instructions copy data into registers from other registers or immediate values.

Where are immediate values stored?

```
MOV  Rd, Op2       // MOVes value of Op2 into Rd
MOV  Rd, #Imm16    // MOVes immediate 16-bit value into Rd

MVN  Rd, Op2       // MOVes complement (Not) of Op2 value
                   // into Rd

MOVT Rd, #Imm16    // MOVes Top: moves a 16-bit constant into
                   // the high-order 16 bits of Rd and leaves
                   // the lower bits unchanged
```

Why is the last instruction useful?

Logic Instructions

These instructions perform binary logic operations on operands, useful for testing conditions, manipulating data, etc.

```
AND  Rd, Rn, Op2  // bitwise AND operation
ORR  Rd, Rn, Op2  // bitwise OR operation
EOR  Rd, Rn, Op2  // bitwise Exclusive OR (XOR) operation
BIC  Rd, Rn, Op2  // BIt Clear: Rd <-- Rn AND NOT(Op2)
```

Shift and Rotate Instructions

Shift and rotate instructions change the positions of bits within a register, moving them left or right.

Note: Last operand can be a register or an immediate value, as with logic operations.

```
LSL  R1, R2, #5   // Logical shift left
LSR  R1, R2, R3   // Logical shift right
ASR  R1, R2, #4   // Arithmetic shift right
```

- Logical ⇒ pad with 0s, Arithmetic ⇒ extend sign bit

```
ROR  R1, R2, #2  // Circular rotate right
```

- Less significant bits (on the right of the register) are moved into the most significant positions (on the left of the register).

Arithmetic Instructions

Addition/subtraction instructions:

```
ADD  R0, R1, R2          // R0 <-- R1 + R2
ADD  R0, R1, #-24        // R0 <-- R1 + (-24)
SUB  R0, R1, #24         // R0 <-- R1 - (24)
ADD  R0, R1, R2, LSL#2   // R0 <-- R1 + R2*4
```

What are the uses of LSL in this case?

Multiply instruction

```
MUL  R2, R3, R4          // R2 <-- R3 * R4
```

Multiply-accumulate instruction

```
MLA  R2, R3, R4, R5      // R2 <-- (R3 * R4) + R5
```

These multiply instructions only return the 32 least significant bits.

There are other, more complex arithmetic instructions; they are not covered in this course.

Arithmetic Instructions

What about division?

```
UDIV  R0, R1, R2          // R0 <-- R1 / R2, R1 and R2 unsigned
SDIV  R0, R1, R2          // R0 <-- R1 / R2, R1 and R2 signed
```

But many processors do not implement *division*.

Division hardware is

- Complex, and therefore costly;
- Slow; and,
- Used infrequently.

Consequently, it is often performed in *software* using an ARM-provided library subroutine (e.g., `aeabi_idiv()`).

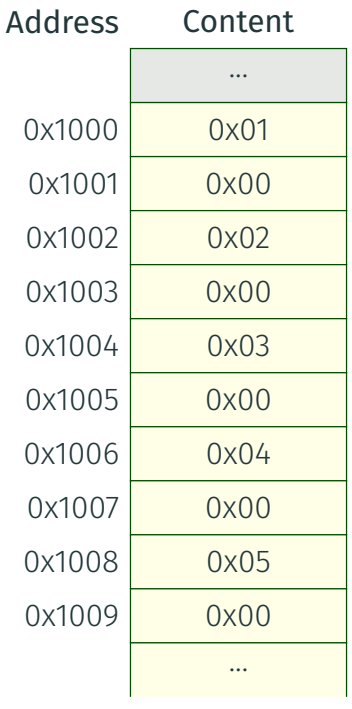
Arrays in C (Review)

```
short arr[5] = {1, 2, 3, 4, 5}
```

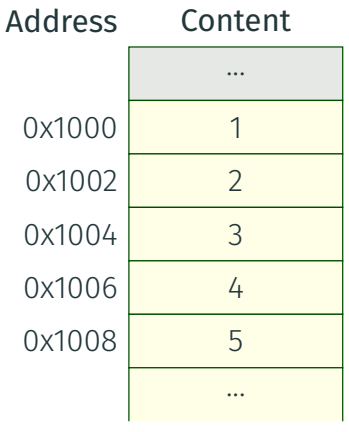
Array elements are allocated one after the other in memory. (Remember endianess!)

For a 1D array, `arr[i]` is stored at address: `&arr[0]+sizeof(TYPE)*i` where

- `&` means *address of*
- `&arr[0]` is the address of the first array element, and base (starting) address of the array
- `sizeof` returns the number of bytes required by `TYPE`
- `sizeof(TYPE)*i` is therefore the offset of element i



Byte view



Half-word view

Array Access Example

Consider the following C code snippet:

```
int arr[8] = {17, 58, 79, 15, ...}; // sizeof(int) = 4 bytes
...
for (int i=0; i<8; i++) {
  v = arr[i];
  ...
  arr[i] = v;
}
```

When reading from an array, we need to:

- Get the base address (`&arr`);
- Multiply the index by the element size (`i*4`) to get the offset;
- Add to calculate the address of the element; and, then, finally
- Access memory!

```c
int arr[8] = {17, 58, 79, 15, ...}; // sizeof(int) = 4 bytes
...
for (int i=0; i<8; i++) {
  v = arr[i];
  ...
  arr[i] = v;
}
```

To access `arr` we need an instruction that can read from memory:

```
LDR  Rd, [Rn]     // Rd <-- Mem[Rn], Rn = address in bytes
```

Our C code is implemented in part with the following assembly:

```
// R0 = variable i
// R1 = base address of arr (&arr)
MOV  R2, #4      // R2 = 4
MUL  R2, R0, R2 // R2 = i*4 -- calculate offset for index i
ADD  R3, R1, R2 // R3 = arr + i*4 -- absolute address of arr[i]
LDR  R4, [R3]   // R4 = arr[i] -- R4 <-- Mem[R1+i*4]
```

```
int arr[8] = {17, 58, 79, 15, ...}; // sizeof(int) = 4 bytes
...
for (int i=0; i<8; i++) {
  v = arr[i];
  ...
  arr[i] = v;
}
```

Address	Content
	...
0x0100	MOV R2,#4
0x0104	MUL R2,R0,R2
0x0108	ADD R3,R1,R2
0x010C	LDR R4,[R3]
	...
0x1000	17
0x1004	58
0x1008	79
0x100C	15
	...

Assume the base address of `arr` is `0x1000` and `i=3`. After execution of the load:

Registers

R0	0x00000003
R1	0x00001000
R2	0x0000000C
R3	0x0000100C
R4	0x0000000F

Load and Store Instructions

Memory accesses commonly* access words and take the form of:

```
LDR  Rd, <EA>  // Rd <-- Mem[EA]; reads a 32-bit word
STR  Rm, <EA>  // Mem[EA] <-- Rn; writes a 32-bit word
```

Loads and stores do not generally specify a memory address explicitly; instead, they compute an effective address (EA) from a base address and an offset.

Effective Address Calculation

$EA = base + offset$

Calculating an EA is very convenient for implementing common program structures: *e.g.*, loops and arrays; and, complex objects.

* *Other load and store instructions access bytes or half words, doubles, or multiple words, and manipulate addresses in more complex ways.*

Effective Address Calculation

- The base address is always stored in a register (Rn)
- There are three kinds of offset:
  - Immediate: a 12-bit number that is added to or subtracted from the base address
  - Index register: the offset is stored in a register (Rm)
  - Scaled index register: the value in the index register is shifted by a specified immediate value, then added to or subtracted from the base address

Methods for Calculating the Effective Address

Name	Assembler syntax	Address generation
register indirect	`[Rn]`	EA = Rn
immediate offset	`[Rn, #offset]`	EA = Rn + offset
offset in Rm	`[Rn, ± Rm, shift]`	EA = Rn ± shifted(Rm)

Back to our Example

```
...
v = arr[i];
...
```

Immediate (with #0): $EA = R3$

```
// R0 = variable i
// R1 = base address of arr (&arr)
MOV R2, #4      // R2 <-- 4
MUL R2, R0, R2  // R2 <-- i*4 -- calculate offset for index i
ADD R3, R1, R2  // R3 <-- arr + i*4 -- absolute address of arr[i]
LDR R4, [R3, 0] // R4 <-- Mem[R3]
```

Index: $EA = R1 + R2$

```
MOV  R2, #4       // R2 <-- 4
MUL  R2, R0, R2   // R2 <-- i*4
LDR  R4, [R1, R2] // R4 <-- Mem[R1+R2]
```

Scaled Index: $EA = R1 + (R0 << 2) = R1 + (R0 \times 4)$

```
LDR  R4, [R1, R0, LSL#2] // R4 <-- Mem[R1+R0<<2]
```

Store Instructions Calculate EA the Same Way

Here's our C code again, but this time we're copying into `arr`:

```
int arr[8] = {17, 58, 79, 15, ...}; // sizeof(int) = 4 bytes
...
for (int i=0; i<8; i++) {
  v = arr[i];
  ...
  arr[i] = v;
}
```

We have the same options for calculating the effective address as we do for load instructions. *E.g.,*:

Scaled Index: $EA = R1 + (R0 << 2) = R1 + (R0 \times 4)$

```
// R0 = variable i
// R1 = base address of arr (&arr)
// R4 = v
STR  R4, [R1, R0, LSL#2] // Mem[R1+R0<<2] <-- R4
```

Checkpoint

For each instruction below, calculate the EA (Effective Address) given the following register content:

```
R2 = 0x1A4DDA38
R6 = 0x10004008
R8 = 0x10004000
R10 = 0x00000002
```

```
LDR R2, [R6, #-4]

LDR R2, [R6, #0x200]

STR R2, [R6, -R8]

STR R2, [R8]

LDR R2, [R8, R10, LSL#3]
```

Pointers in C (Review)

- A pointer (`int *ptr;`) is an address
- You can perform pointer arithmetic to change the address
  - *E.g.*, `ptr = ptr+2;`
  - Also, pre-increment (`++ptr`), and post-increment (`ptr++`)
- You can dereference a pointer (`*ptr`) to access the data at the address

In C, you declare that a variable is a pointer with `*`

```
int *p;     // p is a pointer to an integer
            // i.e. p is the memory address of a 32-bit variable
            // since p contains an address, it is also 32-bits
            // NB that "int* p;" and "int * p;" also do the same thing
int x;
int a[5] = {20, 35, 0, 42, 12};

p = &a[3]; // the address of the 4th element of arr is stored in p

x = *p;     // here, * means indirection (the value addressed by p)
            // it's tricky! C uses * to mean different things in context!
```

What is the value stored in `x`?

If in C we dereference a pointer to access the value at its address:

```
x = *p;
```

This is accomplished with the following assembly:

```
LDR R0, p       // Load the value of p (&arr[3]) into R0
LDR R1, [R0]    // R1 <-- Mem[R0]
STR R1, x       // x <-- R1
```

Why is it important to know the pointer type?

```
int *p;
```

Because we can do arithmetic on the pointer:

```
p = 0x1000;
```

What is `p+1`?

```
int arr[8] = {56, 26, 88, 45, -45, 77, 98, 13};
print(arr);
print(&arr[1]);

int *ptr = &arr[1];
print(ptr);
print(*ptr);

print(ptr+2);
print(*(ptr+2));

print(ptr++);
print(ptr);

print(++ptr);
print(ptr);

print(*(ptr++));
print(*(++ptr));
```

Address	Content
	...
0x1000	56
0x1004	26
0x1008	88
0x100C	45
0x1010	-45
0x1014	77
0x1018	98
0x101C	13
	...

Assuming **`arr`** starts at address **`0x1000`**, what is printed by this C code?

Pointers in Assembly

C code without pointers:

```
int arr[8] = ...;
for (int i=0; i<8; i++) {
  v = arr[i];
  ...
}
```

Loop body in assembly:

```
// R0 = i
// R1 = base address of arr
// R2 = v
LDR R2,[R1,R0,LSL#2] // v=arr[i]
ADD R0,R0,#1         // i++
```

C code with pointers:

```
int arr[8] = ...;
int* ptr = arr;
while (ptr<(arr+8)) {
  v = *(ptr++);
  ...
}
```

Loop body in assembly:

```
// R0 = ptr
// R1 = v
LDR R1, [R0]    // v = *ptr
ADD R0, R0, #4  // ptr=ptr+4
```

Using a pointer instead of `arr[i]` uses one less register in assembly! This is *good for performance*. A good compiler will automatically transform code to use pointers.

Post/Pre-indexed Addressing Mode

ARM includes methods for automatically updating addresses after memory accesses, improving performance.

Recall register indirect addressing:

```
// R0 = ptr
// R1 = v
LDR R1, [R0]    // v = *ptr
ADD R0, R0, #4  // ptr=ptr+4
```

Post-indexed addressing performs the access then updates

```
LDR R1,[R0],#4 // v = *(ptr++)
```

Pre-indexed addressing updates (!) then performs the access

```
LDR R1,[R0,#4]! // v = *(++ptr)
```

Using one instruction to read memory and increment the pointer:

- saves time (fewer instructions are executed),
- saves energy (fewer instructions are read from memory), and
- reduces system costs (less program memory is needed).

Post-indexed addressing performs the access then updates

```
LDR R1,[R0],#4 // v = *(ptr++)
```

Pre-indexed addressing updates (!) then performs the access

```
LDR R1,[R0,#4]! // v = *(++ptr)
```

Assuming R0=0x1008 before the LDR instruction executes, what's the content of R0 and R1 after the instruction executes?

Address	Content
	...
0x1000	56
0x1004	26
0x1008	88
0x100C	45
0x1010	-45
	...

Load/Store Addressing Mode Summary

Name	Assembler Syntax	Address Generation
Register indirect:	`[Rn]`	Address = `Rn`
Offset:		
immediate offset	`[Rn,#offset]`	Address = `Rn` + offset
offset in `Rm`	`[Rn,±Rm,shift]`	Address = `Rn` ± shifted(`Rm`)
Pre-indexed:		
immediate offset	`[Rn,#offset]!`	Address = `Rn` + offset `Rn` ← Address
offset in `Rm`	`[Rn,±Rm,shift]!`	Address = `Rn` ± shifted(`Rm`) `Rn` ← Address
Post-indexed:		
immediate offset	`[Rn],#offset`	Address = `Rn` `Rn` ← `Rn` + offset
offset in `Rm`	`[Rn],±Rm,shift`	Address = `Rn` `Rn` ← `Rn` ± shifted(`Rm`)

- offset = a signed number (12 bits)
- shift = `direction # integer`
  where `direction` is `LSL` for left shift or `LSR` for right shift,
  and `integer` is a 5-bit unsigned number specifying the shift amount

Loading and Storing Byte and Half-words

Dedicated instructions load/store values smaller than a word:

LDRB (Load Register Byte) – zero padded to 32 bits
LDRH (Load Register Halfword) – zero padded to 32 bits

LDRSB (Load Register Signed Byte) – sign extended to 32 bits
LDRSH (Load Register Signed Halfword) – sign extended to 32 bits

STRB (Store Register Byte) – stores low byte of Rd
STRH (Store Register Halfword) – Store the low halfword of Rd

Loading and Storing Multiple Words

`LDM` and `STM` load and store blocks of words in consecutive memory addresses into multiple registers.

`STM`: registers are accessed in order from largest-to-smallest index (`R15..R0`)

`LDM`: registers are accessed in order from smallest to largest index (`R0..R15`)

To determine the direction in which memory addresses are computed, you must use one of the following suffixes for the mnemonic to determine how to update the address:

- IA – Increment After the transfer (default)
- IB – Increment Before the transfer
- DA – Decrement After the transfer
- DB – Decrement Before the transfer

Registers need not be consecutive, *e.g.,*: `LDMIA R8, {R0,R2,R9}`.

Example:

```
LDMIA R3!, {R4, R6-R8, R10}
```

R4 ← Mem[R3]

R6 ← Mem[R3 + 4]

R7 ← Mem[R3 + 8]

R8 ← Mem[R3 + 12]

R10 ← Mem[R3 + 16]

R3 ← R3 + 20 // increment after

PC-relative Addressing

- The PC can be used as the base register to access memory locations in terms of their distance relative to PC+8
  - The processor updates PC ← PC+4, and then fetches the next instruction at that address, which starts executing before the current instruction is finished, so it also increments PC by 4
  - This is called pipelining (covered later)
- PC-relative addressing is used when accessing variable declared statically

Address	Content
	…
0x0FF0	96
0x0FF4	-8
0x0FF8	78
0x0FFC	26
0x1000	`LDR R0, [PC,#-16]`
	…

What's the content of R0 after executing this instruction?

```
LDR R0, [PC, #-16]
```

The SP may be used in a similar way to access data on the stack (more on this later, too).

Assembler Directives

We are almost ready to write out first assembly language program!

The assembler also accepts commands about how it should assemble your program. These are not machine instructions and are never translated to executable machine language.

Some common ones (see the Altera documentation for more):

```
.global symbol     // makes symbol visible outside object file
.word expression   // allocates a 32-bit variable in memory
.equ name, value   // name is replaced with value in this file
.text              // marks the beginning of the code
.end               // marks the end of the code
```

- Text section = where code goes
- Data section = where data goes (*everything except code*)

See more examples of GNU ARM assembly directives.

Loading 32-bit Constants into Registers

We often need a way to load large constant values into registers, *e.g.*, 32-bit addresses. The assembler uses a pseudo-instruction to do this.

```
LDR Rd, =value  // pseudo-instruction: is it a load? a mov?
```

- If the value fits within the range allowed in a `MOV` instruction, the assembler will produce a `MOV` instruction
- Otherwise, the assembler places the constant `value` into a literal pool in memory, in the data section, where it can be read at runtime:

```
LDR Rd, [PC, #offset]
```

where `Mem[PC + offset]` = value.

Example of 32-bit Constants (and our first programs!)

Loading a small constant:

```
.global _start
.text
_start: LDR R0, =0x00000020
.end
```

address	content	code
0x00000000	0xE3A00020	MOV R0, #32

Loading a large constant:

```
.global _start
.text
_start: LDR R0, =0xF0F0F0F0
.end
```

address	content	code
0x00000000	0xE51F0004	LDR R0, [PC, #-4]
0x00000004	0xF0F0F0F0	.word 0xF0F0F0F0

Declaring and initializing a variable, and defining expressions:

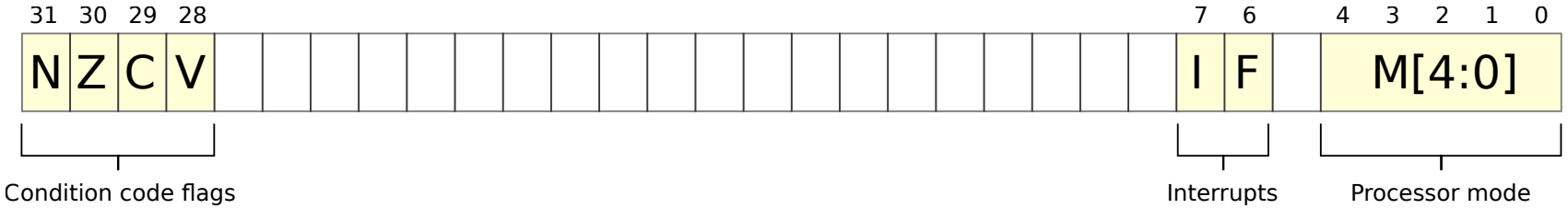
```
.global _start
n: .word 7
.equ m, 0x12
.equ o, 0x1234
_start:
  LDR R0, n    // R0 <-- Mem[n]
  LDR R1, =m   // R1 <-- m
  LDR R2, =o   // R2 <-- o
```

address	content	code
0x00000000	0x00000007	.word 7
0x00000004	0xE54F000c	LDR R0, [PC, #-12]
0x00000008	0xE3A01012	MOV R1, #18
0x0000000C	0xE51F2004	LDR R2, [PC, #-4]
0x00000010	0x00001234	.word 0x00001234

- `LDR R0,n` is a real instruction where the label `n = PC-12`
- `LDR R1,=m` and `LDR R2,=o` are pseudo-instructions

What values are in each register after execution?

Current Program Status Register (CPSR)



- Condition code flag bits are set to 1 when the condition is true
  - N = Negative, Z = Zero, C = Carry, V = Overflow
- Interrupt flags
  - I = IRQ mask bit, F = FRQ (Fast interrupt) mask bit
- Processor mode
  - 10000 = User (most of user code)
  - 10001 = Serving fast interrupt (when dealing with I/O)
  - 10010 = Serving normal interrupt (when dealing with I/O)
  - 10011 = Supervisor (used by the Operating System)

CPSR is not a general-purpose register

Special instructions modify the CPSR, directly or as a side-effect, while others will behave differently depending on CPSR state.

Condition Codes

Combinations of condition code flags are used to determine if the result of an instruction satisfies a particular inequality.

Suffix	Meaning	CPSR Flags
EQ	EQual(zero)	Z=1
NE	Not Equal (nonzero)	Z=0
CS/HS	Carry Set/ unsigned Higher or Same	C=1
CC/LO	Carry Clear / unsigned Lower	C=0
MI	MInus (negative)	N=1
PL	PLus (positive or zero)	N=0
VS	oVerflow Set	V=1
VC	oVerflow Clear	V=0
HI	unsigned Higher	C=1 AND Z=0
LS	unsigned Lower or Same	C=0 OR Z=1
GE	signed Greater or Equal	N=V
LT	signed Less Than	N!=V
GT	signed Greater Than	Z=0 AND (N=V)
LE	signed Less or Equal	Z=1 OR (N!=V)
AL	ALways executed	None tested

Branch Instructions

Branch instructions read the condition code flags to determine whether or to jump to a label, or continue with the next instruction.

```
B{cond} LABEL
```

- The condition **cond** specifies a test of the condition code bits
- If the condition is true, the next instruction executed will be at address **LABEL**, the target
- If the condition is false, the processor simply executes the next instruction (fall-through)

Branch instructions enable control flow

Branch instructions are essential for control flow operations in software: *e.g.*, if/else, loops, function calls, etc.

Test & Compare Instructions

Some instructions are designed specifically to set condition flags:

```
TST  Rs, Op2
```

Zero flag (Z) set to result of `AND(Rs, Op2)`

```
TEQ  Rs, Op2
```

Zero flag (Z) set to result of `XOR(Rs, Op2)`

```
CMP  Rs, Op2
```

Condition code flags set to result of `Rs - Op2` (`Rs` unchanged)

```
CMN  Rs, Op2
```

Condition code flags set to result of `Rs + Op2` (`Rs` unchanged)

Example

C code:

```
if (x>3)
  y = 7;
else
  y = 13;
```

Corresponding ARM assembly:

```
      LDR R0, X    // R0 <-- Mem[X]
      CMP R0, #3   // R0-#3, only update CPSR
      BLE ELSE     // if R0-#3<=0 then branch
      MOV R1, #7   // ** if code **
      B   END      // branch to END
ELSE: MOV R1, #13  // ** else code **
END:  STR R1, Y    // Mem[Y] <-- R1
```

As an exercise, determine the contents of each register and CPSR after each instruction, assuming:

1) x = 6,
2) x = 2, and
3) x = 3

Conditional Execution

Branch instructions are executed when the stated condition is true. Most ARM instructions can be executed conditionally, too.

Instruction format: `OP{S}{cond} Rd, Rn, Op2`

```
if (x>3)
  y = 7;
else
  y = 13;
```

```
LDR   R0, X
CMP   R0, #3  // set flags
MOVGT R1, #7  // if R0-3 >  0
MOVLE R1, #13 // if R0-3 <= 0
STR   R1, Y
```

If the condition is true, then the instruction executes, otherwise the instruction has no effect. This can save some branches, resulting in compact and fast code.

This is a pretty advanced and ARM-specific technique. For now, thinking in terms of branches keeps things simple.

Dot Product

The dot product of two vectors A and B is defined as:

$$\sum_{i=0}^{n-1} A(i) \cdot B(i)$$

C program for the dot product of two vector of six integers:

```
void main() {
  int n = 6;
  int vectorA[6] = {5, 3, -6, 19, 8, 12};
  int vectorB[6] = {2, 14, -3, 2, -5, 36};
  int dotP;
  int i;

  dotP = 0;
  for (i = 0; i<n; i++)
    dotP += vectorA[i] * vectorB[i];

  printf("Dot product = %d\n", dotP);
}
```

C variable declarations:

```
int n = 6;
int vectorA[6] = {5, 3, -6, 19, 8, 12};
int vectorB[6] = {2, 14, -3, 2, -5, 36};
int dotP;
int i;
```

Assembly memory allocation:

```
n:       .word 6
vectorA: .word 5,3,-6,19,8,12
vectorB: .word 2,14,-3,2,-5,36
dotP:    .space 4
// i will be stored in a register, no memory allocation needed
```

- `.word a, b, c, ...`
  allocate storage for 1 or more words (4 bytes each) and initialize with the values a, b, c, ...
- `.space 4`
  allocate 4 bytes without initialization
- `n`, `vectorA`, ... are addresses (*labels*) corresponding to the start of the allocated space

The for loop expands to a number of initialization instructions and other code that is repeated once each iteration.

```
dotP = 0;
for (i = 0; i<n; i++)
  dotP += vectorA[i] * vectorB[i];
```

```
 MOV R3, #0          // register R3 will accumulate the product

 LDR R0, =vectorA    // R0 <-- vectorA base address (pseudo-instruction)
 LDR R1, =vectorB    // R1 <-- vectorB base address (pseudo-instruction)
 LDR R2, n           // R2 <-- Mem[n] = 6

 MOV R6, #0          // initialize iteration variable i

LOOP:
 CMP R6, R2          // do i-n and set flags accordingly
 BGE END             // we're done if i-n >= 0 (if i >= n)
 LDR R4, [R0], #4    // get vectorA[i]; post-index increments R0 after
 LDR R5, [R1], #4    // get vectorB[i]; post-index increments R1 after
 MLA R3,R4,R5,R3     // R3 <-- (R4*R5)+R3
 ADD R6,R6,#1        // i++
 B   LOOP

END:
 STR R3, dotP        // Mem[dotP] <-- R3
```

A more efficient approach uses **SUBS**:

```
dotP = 0;
i = n;
do { // assumes there is at least one element in each array
  dotP += vectorA[i] * vectorB[i];
  i--;
} while (i>0)
```

```
 MOV R3, #0          // register R3 will accumulate the product

 LDR R0, =vectorA    // R0 = vectorA base address (pseudo-instruction)
 LDR R1, =vectorB    // R1 = vectorB base address (pseudo-instruction)
 LDR R2, n           // R2=6 (R2 is i this time)

LOOP:
 LDR  R4, [R0], #4   // get vectorA[i]; post-index increments R0 after
 LDR  R5, [R1], #4   // get vectorB[i]; post-index increments R1 after
 MLA  R3,R4,R5,R3    // R3 = (R4*R5)+R3
 SUBS R2,R2,#1       // i-- and set condition flags
 BGT  LOOP           // we're not done if i>0

 STR R3, dotP
```

- One less register used
- 5 vs 7 instructions in the loop body

Last bit, printing the result:

```
printf(``Dot product = %d\n'', dotP);
```

We have to call a library subroutine to print the results. This usually requires an operating system to print information on a terminal, or direct access to an I/O device in assembly (*e.g.*, a screen). We will see that in another lecture.

Full dot product code in ARM assembly

```
.global _start // tells the assembler/linker where to start execution

n:        .word 6
vectorA:  .word 5,3,-6,19,8,12
vectorB:  .word 2,14,-3,2,-5,36
dotP:     .space 4

_start:
 MOV R3, #0           // register R3 will accumulate the product
 LDR R0, =vectorA     // R0 = vectorA base address (pseudo-instruction)
 LDR R1, =vectorB     // R1 = vectorB base address (pseudo-instruction)
 LDR R2, n            // R2=6 (R2 is our loop iteration variable i)

LOOP:
 LDR  R4, [R0], #4    // get vectorA[i]; post-index increments R0 after
 LDR  R5, [R1], #4    // get vectorB[i]; post-index increments R1 after
 MLA  R3,R4,R5,R3     // R3 = (R4*R5)+R3
 SUBS R2,R2,#1        // i-- and set condition flags
 BGT  LOOP            // we're not done if i>0

 STR R3, dotP         // save our result in memory

STOP:
  B    STOP           // infinite loop once we're done
```

Subroutines

It is typical programming practice to reuse blocks of code in a subroutine (*i.e.*, procedure, function, method) that can be called from many places in a program.

```
int add3(int a, int b, int c) {
  return a + b + c;
}

void main() {
  int sum = 0;

  sum += add3(1, 2, 3);
  sum += 10;
  sum += add3(10, 20, 30);

  printf("Sum = %d\n", sum);
}
```

Requirements for calling subroutines:

- We should be able to call a subroutine from anywhere in our program, *i.e.*, change the PC so that the routine is executed
- A subroutine must be able to return, *i.e.*, change the PC so that execution continues immediately after the point where it was called
- We should be able to pass parameters (or arguments) that may take different values across different calls
- A subroutine must be able to return a value

```
int add3(int a,int b,int c)
{
  return a + b + c;
}

void main() {
  int sum = 0;

  sum += add3(1, 2, 3);
  sum += 10;
  sum += add3(10, 20, 30);

  printf("Sum = %d\n", sum);
}
```

Calling and Returning

A subroutine call is implemented with the Branch and Link instruction BL that stores the address of the next instruction (return address) in the link register LR (R14).

```
BL addr  // LR <-- PC +4; PC <-- addr
```

To return, branch to the address stored in the link register with the BX instruction (branches to the address in a register).

```
BX Rn    // PC <-- Rn
```

C code:

```
boo() {
  coo();
  ...
}
coo() {
  ...
  return;
}
```

ARM assembly:

```
boo:  BL coo  // LR <-- PC +4; PC <-- coo
      ...
coo:  ...
      BX LR  // PC <-- LR
```

Nested Subroutine Calls

```
boo() {
      coo();
B1:   doo();
B2:
return;
}
coo() {
     doo();
C:   return;
}
doo() {
     return;
}
```

- These calls are nested: boo calls coo, coo calls doo
- If we save return addresses in LR, calling doo from coo overwrites the return address back to boo!
- doo() is called from two different places, and is expected to return to different places for each call
- How do we remember the return addresses for each call, in the correct order? (*I.e.*, the reverse call order.)

boo calls **coo**	save B1
coo calls **doo**	save C
doo returns to **coo**	PC ← C
coo returns to **boo**	PC ← B1
boo calls **doo**	save B2
doo returns to **boo**	PC ← B2

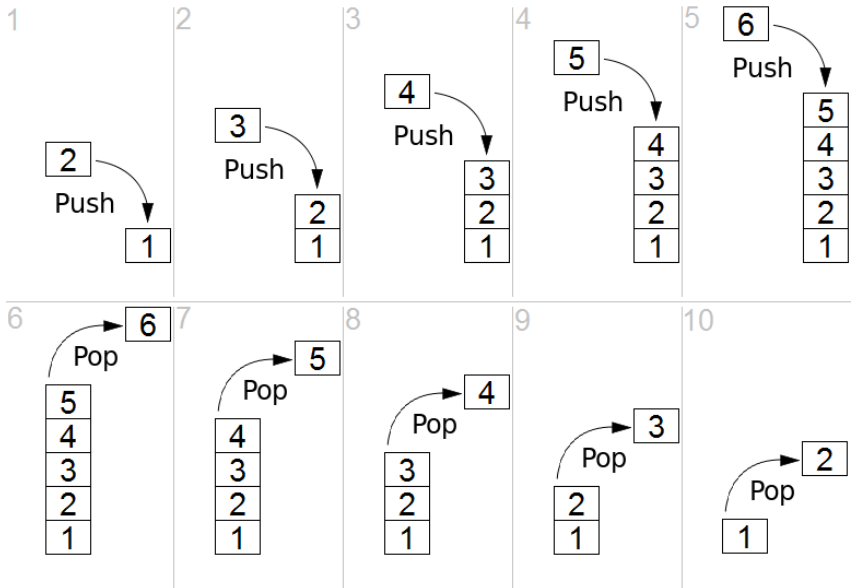
Which data structure shall we use to save these addresses?

We need a way to recall return addresses (and later, other things) in the opposite order they were saved.
We will use a Last-in-First-out (LIFO) data structure called a stack! The stack is saved in main memory, and accessed with special load and store instructions.

source: Mk2010 / CC 4.0 BY-SA

Stack Operations

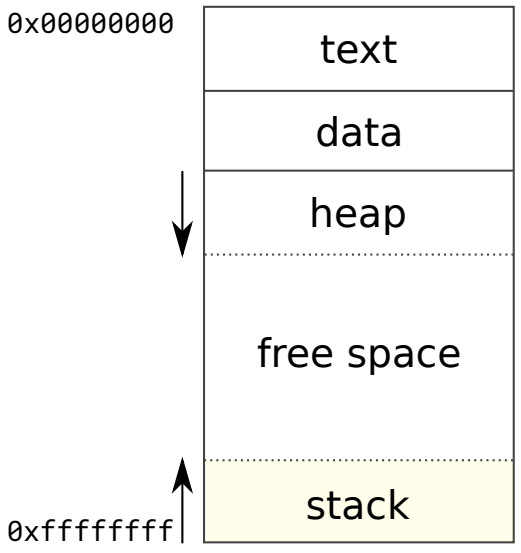
- `push(value)`: adds new item value to top of the stack (TOS)
- `value = pop()`: returns and removes the top element
- `value = peek(distance)`: returns (but does not remove) the value of an element at a distance relative to TOS; `peek(0)` returns the element at the TOS



source: Maxtremus / CC0

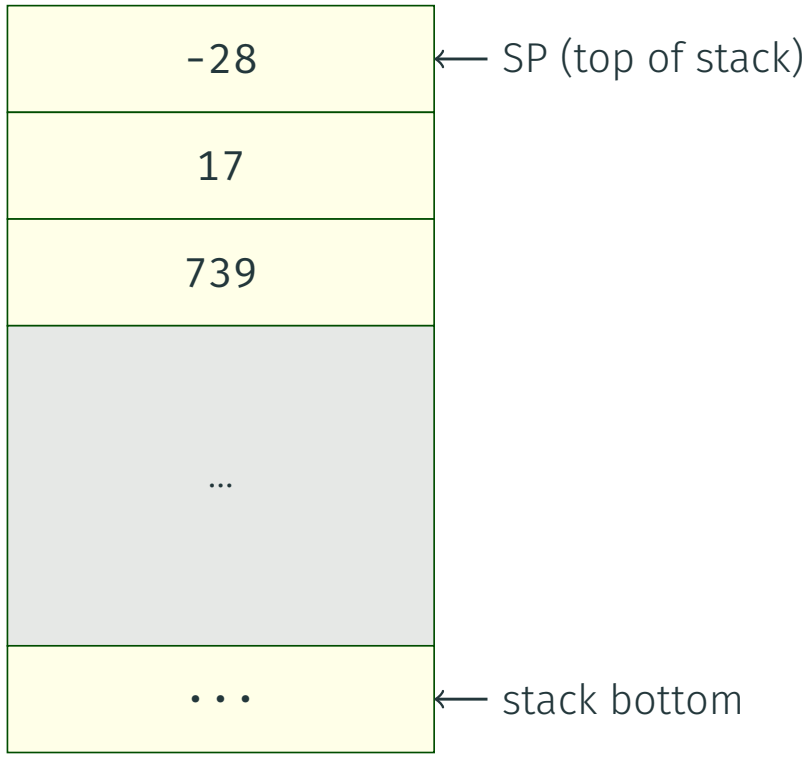
ARM Memory Layout

- Recall: text is where compiled code goes
- Recall: data is where compile-time statically allocated data goes
- The size of text and data sections are fixed at compile-time
- The heap is where dynamically allocated (*e.g.*, using new or malloc) data goes
- The heap starts at lower addresses and grows "downward" toward higher addresses
- The bottom of the stack is at a fixed address and the top of stack grows "upward", towards lower memory addresses



The Stack in ARM

- The stack is used to support subroutines: saving return addresses, function arguments, etc
- The data elements on the stack are always* words; memory accesses to the stack are always* aligned
- Register **R13** is the stack pointer (**SP**); it points to TOS



* by convention; breaking from convention may break your code

Stack Operations in ARM

Push from Rj

```
STR Rj, [SP, #-4]!
```

SP ← SP - 4

Mem[SP] ← Rj

Pop into Rj

```
LDR Rj, [SP], #4
```

Rj ← Mem[SP]

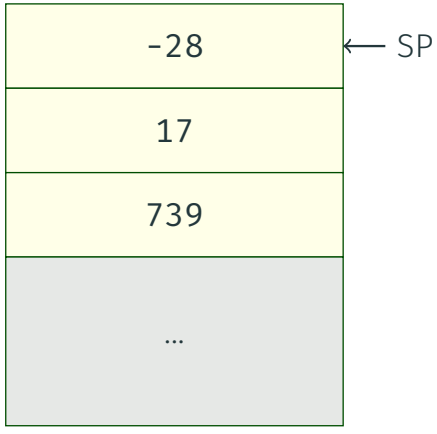
SP ← SP + 4

Peek(i) into Rj

```
LDR Rj, [SP, #const]
```

where const $= i * 4$

Rj ← Mem[SP+const]



Assuming Rj=19, SP=0xFFFFABCC and i=2, what's the content of the stack, register Rj, and SP, after each instruction executes? (consider them separately)

Pushing and Popping Multiple Elements

Often, several elements need to be pushed/popped onto/from the stack, *e.g.*, at the start and end of subroutines.

There are two pseudoinstructions that are useful aliases for STM and LDM (slide 38):

- `PUSH {R1, R3-R5}` is equivalent to
  `STMDB SP!, R1, R3-R5`
  (`R5` is pushed first, and `R1` ends up at the top of the stack)
- `POP {R1, R3-R5}` is equivalent to
  `LDMIA SP!, R1, R3-R5`
  (top of the stack ends up in `R1`)

Nested Subroutine Calls, Revisited

```
main() {
      boo();
A:    ...;
}
boo() {
      push(LR);
      coo();
B1:   doo();
B2:   LR = pop();
      return;
}
coo() {
     push(LR);
     doo();
C:   LR = pop();
     return;
}
doo() {
     return;
}
```

Subroutines that might call another subroutine must follow this convention:

- Before you call a subroutine: push the return address stored in LR onto the stack
- When the subroutine returns: pop the return address off the stack into LR

Action	Stack (TOS on left)	LR
main calls boo		A
boo saves LR	A	A
boo calls coo	A	B1
coo saves LR	B1 A	B1
coo calls doo	B1 A	C
doo returns	B1 A	C
coo restores LR	A	B1
coo returns	A	B1
boo calls doo	A	B2
doo returns	A	B2
boo restores LR		A
boo returns		A

Passing parameters and returning values

For a small number of parameters, the ARM APCS recommends using:

- R0 – R3 (A1 – A4) for passing parameters, and
- R0 (A1) for the return value

```
int add3(int a, int b, int c) {
  return a + b + c;
}
```

```
      MOV  R0, #1
      MOV  R1, #2
      MOV  R2, #3
      PUSH {LR}      // STR LR,[SP,#-4]!; saves return address
      BL   add3
      STR  R0, SUM   // return value is in R0
      POP  {LR}      // LDR LR,[SP],#4; restores return address
      ...

add3: ADD  R0, R0, R1
      ADD  R0, R0, R2
      BX   LR
```

ARM APCS Uses the Callee-save Convention

```
add3:   ADD R0, R0, R1
        ADD R0, R0, R2
        BX  LR
```

- In the previous example, the callee overwrote **R0**, which was OK, since the caller knew that the return value would be in **R0**
- In general, the caller may need the register values after the callee returns, so the rule is a callee is responsible for leaving the registers as it found them

Callee-save convention:

A subroutine should save any* registers it wants to use on the stack and then restore the original values to the registers after it is finished using them.

* The ARM APCS states that argument registers **A1** – **A4** need not be saved, but remember: they might be changed inside of subroutines!

Registers in the ARM Architecture Procedure Call Standard

Most registers are callee-saved: if a subroutine is going to use them, their state must first be saved (on the stack), and later restored (from the stack).

Register	Synonym	Special	Role in the AAPCS
r15		PC	Program counter
r14		LR	Link register
r13		SP	Stack pointer
r12		IP	Intra-procedure scratch register
r11	v8	FP	Frame pointer OR variable register 8
r10	v7		Variable register 7
r9		v6/SB/TR	Platform register
r8	v5		Variable register 5
r7	v4		Variable register 4
r6	v3		Variable register 3
r5	v2		Variable register 2
r4	v1		Variable register 1
r3	a4		Argument / scratch register 4
r2	a3		Argument / scratch register 3
r1	a2		Argument / result / scratch register 2
r0	a1		Argument / result / scratch register 1

Passing Parameters On the Stack

When you have more than four parameters, you can pass four in registers, and the additional ones on the stack. (This is what compilers do, and what the APCS recommends.)

Or, you can pass all parameters and the return value on the stack. Passing parameters in registers will always be faster. Why?

When you want to pass a data structure that does not fit into four words, you must use the stack (for at least part of it). Example:

```
struct largeDataStruct {
  int a;
  int b;
  int c;
  int d;
  int e;
}
```

Let's see how to pass everything on the stack with a program that sums a list of numbers.

```
ARRAY:    .word 6,5,4,3,2,1,14,13,12,11,10,9,8,7   // sum these
N:        .word 14                                 // this many of them
SUM:      .space 4                                 // result goes here
          .global _start
_start:   LDR   A1, =ARRAY     // A1 points to ARRAY
          LDR   A2, N          // A2 contains number of elements to add
          PUSH  {A1, A2, LR}   // push parameters and LR (A1 is TOS)
          BL    listadd        // call subroutine
          LDR   A1, [SP, #0]   // return is at TOS
          STR   A1, SUM        // store it in memory
          ADD   SP, SP, #8     // clear parameters
          POP   {LR}           // restore LR
stop:     B     stop
listadd:  PUSH  {V1-V3}        // callee-save registers listadd uses
          LDR   V1, [SP, #16]  // load param N from stack
          LDR   V2, [SP, #12]  // load param ARRAY from stack
          MOV   A1, #0         // clear R0 (sum)
loop:     LDR   V3, [V2], #4   // get next value from ARRAY
          ADD   A1, A1, V3     // form the partial sum
          SUBS  V1, V1, #1     // decrement loop counter
          BGT   loop
          STR   A1, [SP, #12]  // store sum on stack, replacing ARRAY
          POP   {V1-V3}        // restore registers
          BX    LR
```

Passing by Value, Passing by Reference

Recap from C:

- Passing by value: a copy of the value is passed to the callee. If the copy is modified, *there is no effect on the caller side*.
- Passing by reference: an address in memory where the value is stored is passed. The callee may modify the value.

```
int add3Val(int a) {
  a = a+3;
  return a;
}
void add3Ref(int* a) {
  *a = (*a)+3
}
void main() {
  int i=77;
  int j;

  j = add3Val(i);
  print(i);
  print(j);

  add3Ref(&i);
  print(i);
  print(j);
}
```

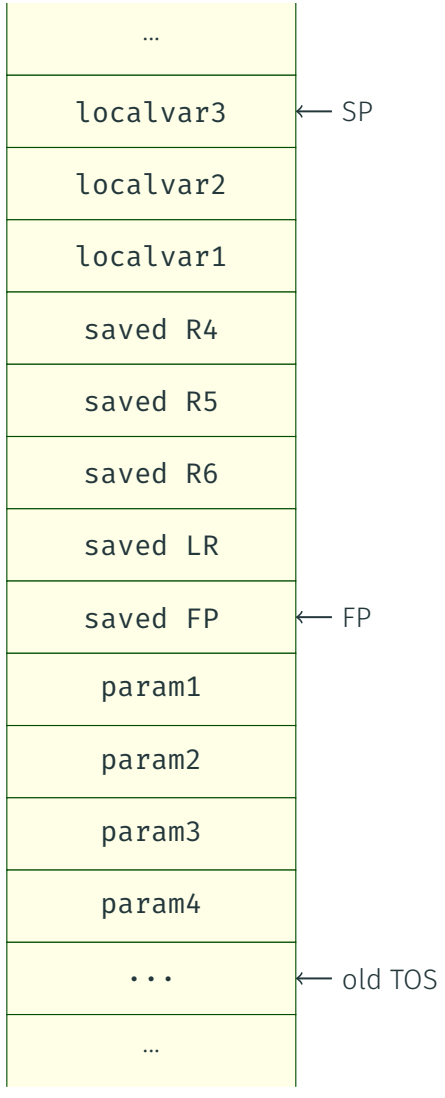
```
ARRAY:    .word 6,5,4,3,2,1,14,13,12,11,10,9,8,7
N:        .word 14
...

LDR   A1, =ARRAY     // A1 points to ARRAY
LDR   A2, N           // A2 contains number of elements to add
PUSH  {A1, A2, LR}    // push parameters and LR (A1 is TOS)
BL    listadd         // call subroutine
```

- The parameter N was passed by value, *i.e.*, the actual value of N (14) was passed to the subroutine; as it was modified in the routine, the value in memory was not changed.
- The parameter ARRAY was passed by reference, *i.e.*, a pointer to the first element of the array was passed; if we'd changed elements, they'd have been changed in memory.

Stack Frame

- The subroutine can* also allocate local variables, only accessible by the subroutine, on the stack.
- Using a frame pointer (`R11`) gives a consistent reference to parameters `[FP, #const]` and local variables `[FP, #-const]`
- When nesting, the stack frame also includes the return address and frame pointer
- `FP` is not strictly required; it is mainly used to make assembly programs easier to write, and to help with the debugger
- `FP` remains constant while in the same subroutine



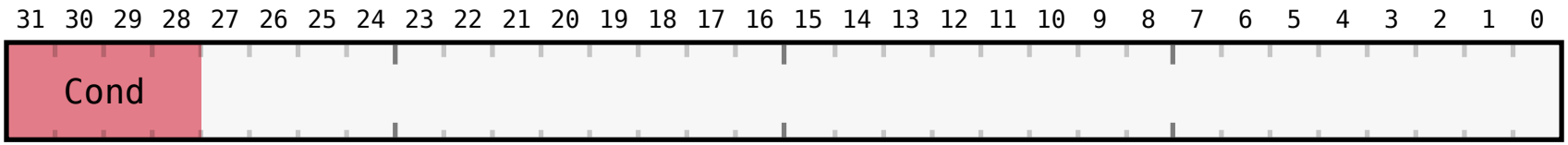
* Most local variables are actually allocated this way, reducing the total memory required by a program.

ARM Assembly vs. Binary

Machine language instruction are encoded as binary, with 32* bits per instruction (ARM ISA is RISC).

The binary representation of an instruction is divided into fields. Each field encodes different information about the instruction.

The general format for most instructions:



source: https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/

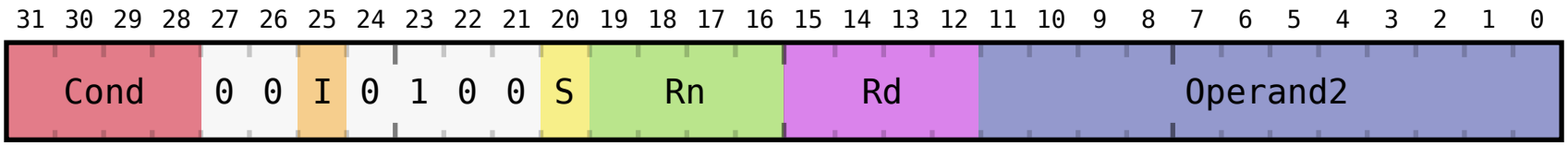
* 16-bit versions are available for many instructions, but such instructions tend to be less flexible.

Condition Field

Cond. field	Suffix	Meaning	CPSR Flags
0000	EQ	EQual(zero)	Z=1
0001	NE	Not Equal (nonzero)	Z=0
0010	CS/HS	Carry Set/ unsigned Higher or Same	C=1
0011	CC/LO	Carry Clear / unsigned Lower	C=0
0100	MI	MInus (negative)	N=1
0101	PL	PLus (positive or zero)	N=0
0110	VS	oVerflow Set	V=1
0111	VC	oVerflow Clear	V=0
1000	HI	unsigned Higher	C=1 AND Z=0
1001	LS	unsigned Lower or Same	C=0 OR Z=1
1010	GE	signed Greater or Equal	N=V
1011	LT	signed Less Than	N!=V
1100	GT	signed Greater Than	Z=0 AND (N=V)
1101	LE	signed Less or Equal	Z=1 OR (N!=V)
1110	AL	ALways executed	None tested

1111 is not used.

Data Processing Instructions Encoding



source: https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/

Examples:

```
ADDGES R1, R2, R3
```

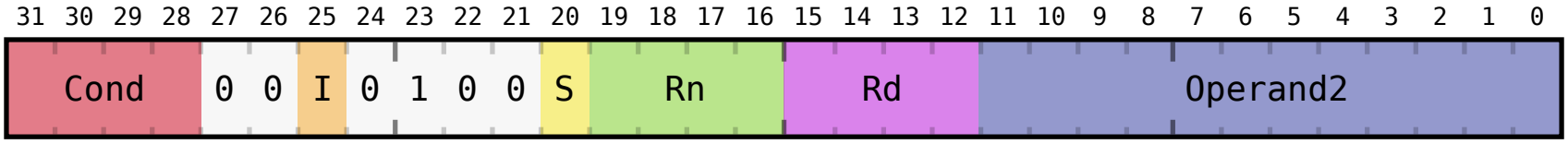
Cond=1010, I=0, S=1, Rn=0010, Rd=0001, Operand2[3-0]=0011

```
ADD R1, R2, #15
```

Cond=1110, I=1, S=0, Rn=0010, Rd=0001, Operand2=000000001111

Why are the register fields 4 bits wide?

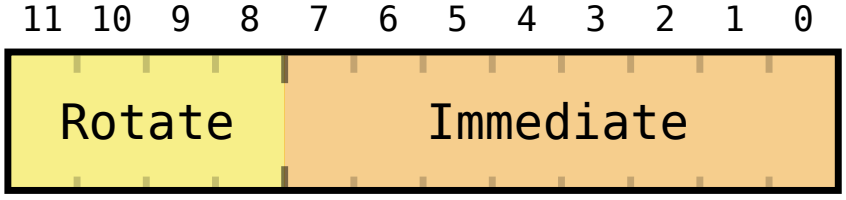
Immediate Value Encoding



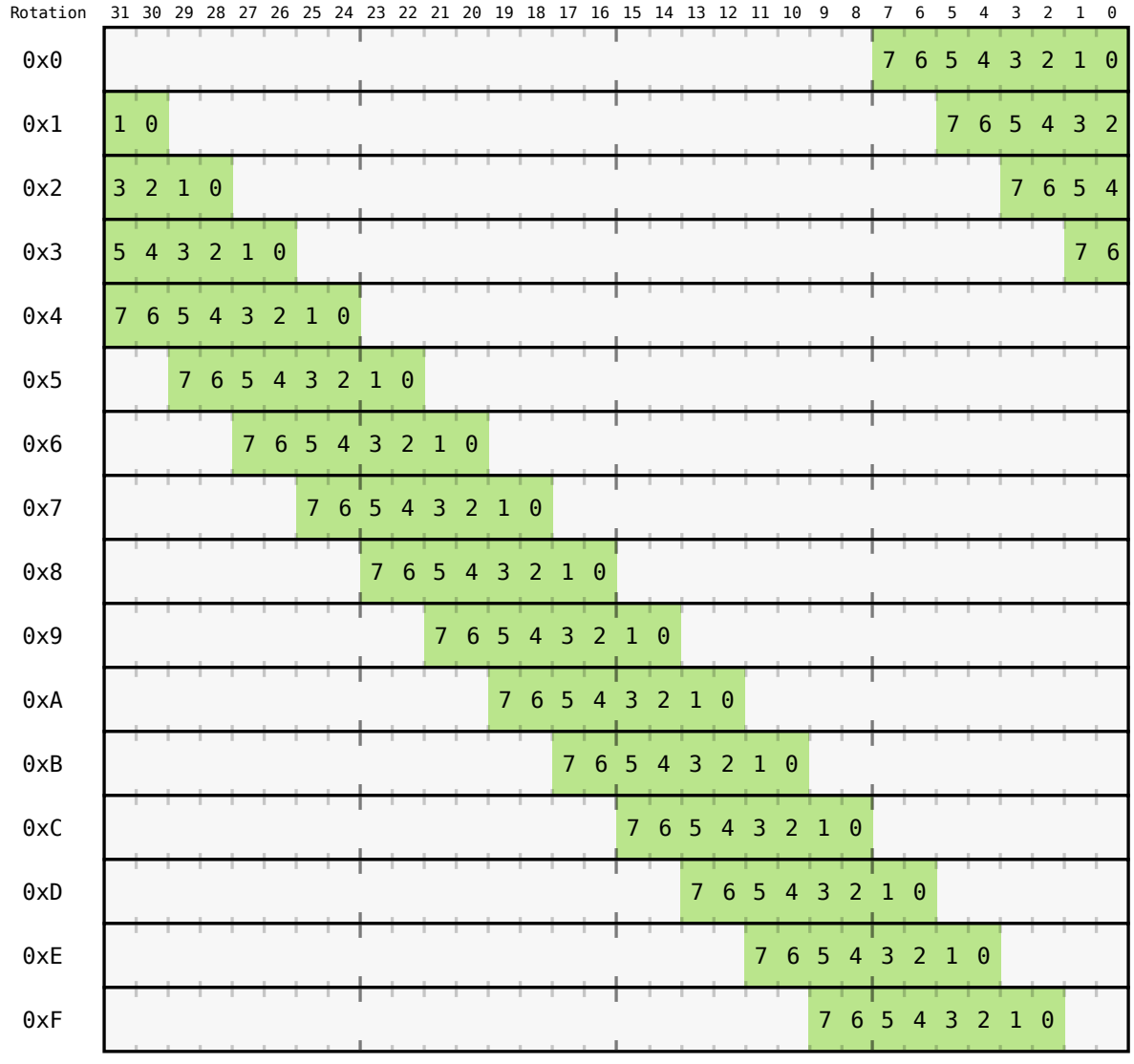
source: https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/

12 bits are available to encode immediate value. However, the largest value is not what you think it might be.

The ARM ISA has a very clever way of generating a lot of useful 32-bit constants: 16 possible rotations of an 8-bit value



source: https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/

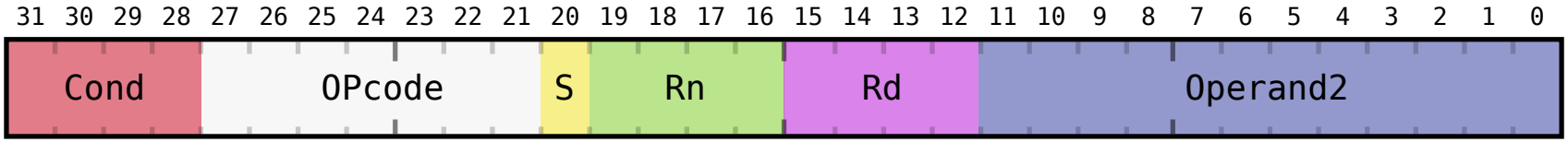


source: https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/

Rotations of an even number of times in a 32-bit word (0, 2, ..., 30)

https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/

Load/Store Instruction Encoding



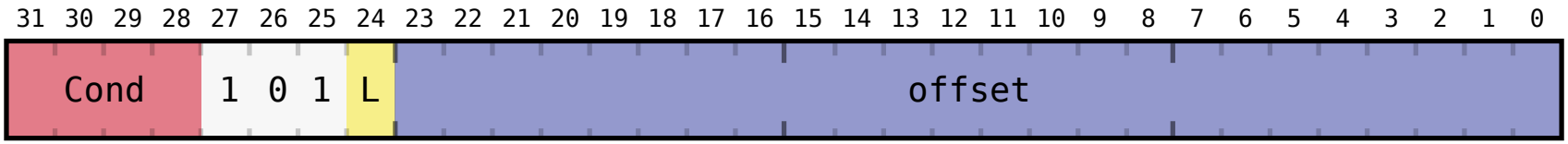
`Rn` is the base address.

`Operand2` is the offset: an immediate value, or register value (four LSBs), or register (four LSBs) and shift amount (five MSBs).

Note that:

- Not every addressing mode is available for every load/store instruction.
- The range of permitted immediate values and the options for scaled registers vary from instruction to instruction.

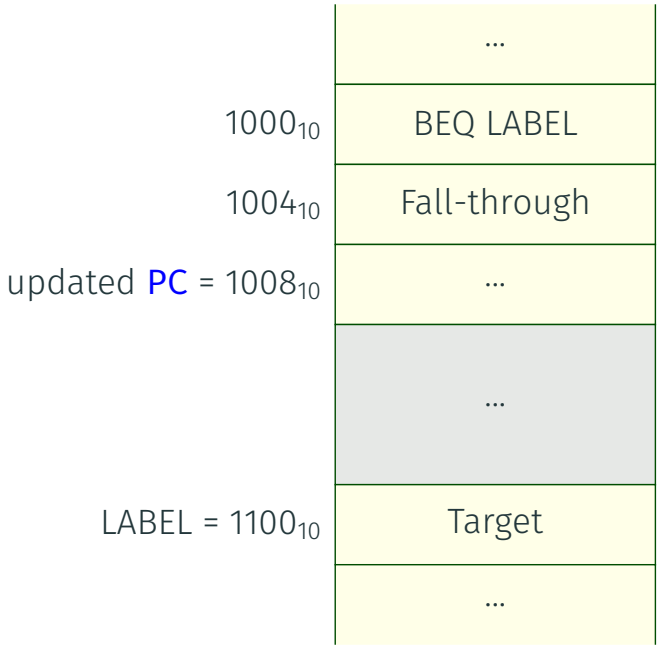
Branch Instruction Encoding



Since the offset field is limited to 24 bits:

- the branch target address is relative to the current value of PC,
- the offset is left-shifted twice (offset is in words, not bytes)

`L=1` is used for the `BL` instruction.



In this example, we want to jump to address 1100_{10} which is 100 bytes away.

The relative offset is 92 bytes (100 – 8)
= 23 words
= `0000 0000 0000 0000 0001 0111`.

The condition field is EQ = `0000`.

Conclusions

This set of lectures has presented the ARM ISA and introduced:

- the major classes of instructions
- the different addressing modes used by memory accesses
- the way ARM branches work
- the way subroutine calls are implemented in assembly with the stack
- the encoding of instructions in binary

The next lecture will:

- look at the software toolchain used to translate high-level languages to machine code
- the role of the operating system software