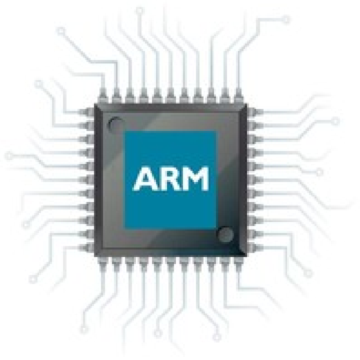
Advanced RISC Machine (ARM)

architecture and assembly language

Simone Aonzo



30$^{\text{th}}$ July 2018

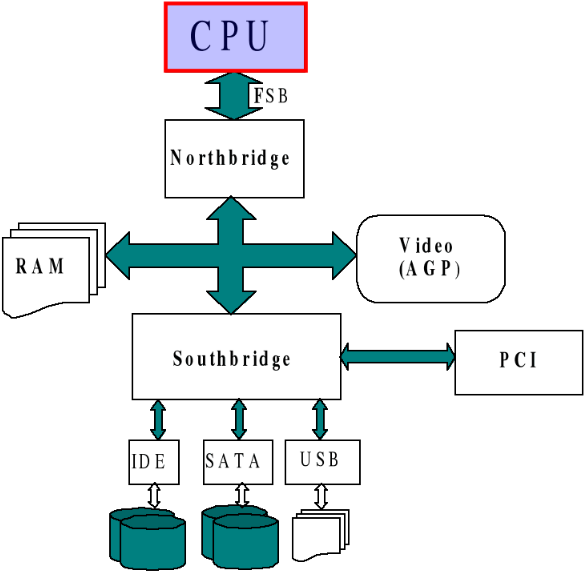
Index

1. Background
2. Advanced RISC Machine architecture
3. ARM assembly language
   - Arithmetic instructions
   - Condition codes
   - Data transfer instructions
   - Control transfer instructions
   - Stack instructions
4. ARM32 Architecture Procedure Calling Standard

Outline

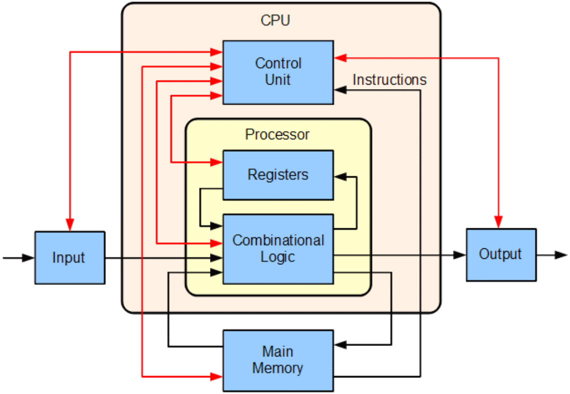
1. Background
2. Advanced RISC Machine architecture
3. ARM assembly language
   - Arithmetic instructions
   - Condition codes
   - Data transfer instructions
   - Control transfer instructions
   - Stack instructions
4. ARM32 Architecture Procedure Calling Standard

PC architecture

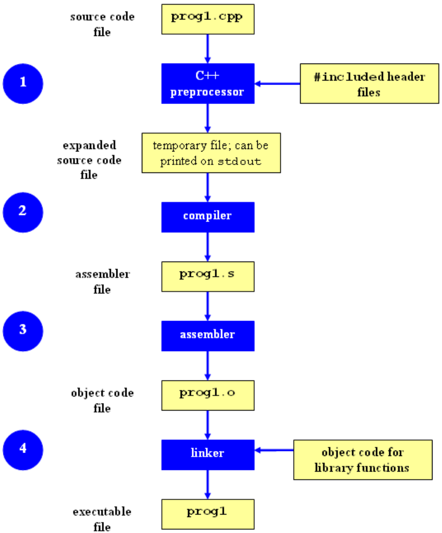


Central Processing Unit

A central processing unit (CPU) is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output operations ***specified by the instructions****.* [Wikipedia]



The compilation process



Executable file

An **executable file** causes a computer *to perform indicated tasks according to encoded instructions* (i.e. machine code instructions for a physical CPU) as opposed to a **data file** that must be parsed by a program to be meaningful.

The ones in most common use are:

- Portable Executable (PE) on Microsoft Windows
- Executable and Linkable Format (ELF) on Linux|Unix
- Mach Object (Mach-O) on macOS and iOS

Machine and assembly languages

We are concerned with two types of languages, assembly languages and machine languages:

- a **machine language** encodes instructions as sequences of 0's and 1's that is what the computer's processor is built to execute;
- when programmers want to dictate the precise instructions that the computer is to perform, they use an **assembly language**, which allows instructions to be written in textual form.
- An **assembler** translates a file containing assembly language code into the corresponding machine language.

Machine and assembly languages example

- Here is a machine language instruction:

  1110 0001 1010 0000 0011 0000 0000 1001

- But a programmer would prefer programming in assembly language, where we would express this using the following line.

  MOV R3, R9

  Then the programmer would use an assembler to translate this into the binary encoding that the processor actually executes.
- When the processor executes this binary sequence, it copies the bits contained in register 9 into register 3.

Instruction Set Architecture

There is not just one machine language:

- a different machine language is designed for each line of processors
- often processors are designed to be compatible with a previous processor, so it follows the same machine language design
- the design of the machine language encoding is called the **instruction set architecture (ISA)**

CISC and RISC

Complex instruction set computer (CISC) is a processor design where single instructions:

- can execute several low-level operations
- are capable of multi-step operations or complex addressing modes within single instructions

Reduced instruction set computer (RISC) is one whose ISA has a set of attributes that:

- allow them to have a lower cycles per instruction than a complex instruction set computer (CISC)
- memory is only accessed through specific instructions, rather than as a part of most instructions (as is the case in CISC)

General concept

A RISC processor has a small set of simple and general instructions, rather than a CISC that has a large set of complex and specialized instructions

CISC vs RISC

CISC

- Emphasis on hardware
- Includes multi-clock complex instructions
- Memory-to-memory $\Rightarrow$ LOAD/STORE incorporated in instructions
- Small code sizes, but high cycles per second
- Transistors used for storing complex instructions

RISC

- Emphasis on software
- Single-clock reduced instruction only
- Register-to-register $\Rightarrow$ LOAD/STORE are independent instructions
- Low cycles per second, but large code sizes
- Spends more transistors on memory registers

The success of RISC

Processors that have a RISC architecture typically require fewer transistors than CISC, which improves:

1. cost
2. power consumption
3. heat dissipation

These characteristics are desirable for portable and battery-powered devices like smartphones and embedded systems.
But also for supercomputers, which consume large amounts of electricity.

Tradeoff

A program for a RISC architecture needs more memory than a CISC one, because a single (slow) instruction in CISC may require two, or more simpler RISC instructions

ISA main varieties

- **x86** (CISC) is handily the most widely recognized
  - 1974: Intel 8080, 8-bit
  - 1978: Intel 8086, 16-bit
  - 1985: Intel 80386, 32-bit (aka **i386** or IA-32)
  - 2003: AMD Opteron, 64-bit (aka **x64**, x86_64, AMD64 or Intel 64)
- **ARM** (RISC)
  - 1985: ARMv1, 32-bit
  - ...
  - 2011: ARMv8-A, 64-bit
- **MIPS** (RISC) - introduced in 1985
  - currently used in embedded systems such as routers
  - in the past for personal, workstation, and server computers
  - but also video game consoles: Nintendo 64, PSX, PS2, PSP
- **PowerPC** (RISC) - introduced in 1992
  - Apple's Macintosh computers until 2006, then switched to the x86
  - gaming consoles like Wii, PS3, and XBox 360
- and so many other...

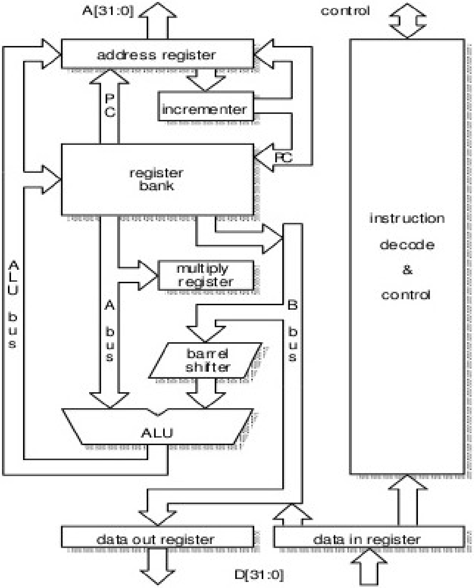
Outline

1. Background
2. Advanced RISC Machine architecture
3. ARM assembly language
   - Arithmetic instructions
   - Condition codes
   - Data transfer instructions
   - Control transfer instructions
   - Stack instructions
4. ARM32 Architecture Procedure Calling Standard

ARM Advanced RISC Machine

1. a family of RISC processors
   - load/store based architecture
   - single-cycle instruction execution
   - small instruction set
   - fixed instruction size
2. an assembly language

ARM Architecture



ARM instruction set

The ARM processor has 3 instruction set:

1. traditional ARM (32/64 bit instructions)
2. more condensed Thumb (16 bit instructions)
3. Jazelle (native execution of Java bytecode)

Why Thumb?

- to reduce code size $\Rightarrow$
  - reducing the total amount of memory required
  - narrowing the data bus to just 16 bits
- Thumb $\subset$ ARM i.e. different set represented by the same language
- Thumb $\oplus$ ARM i.e. only one set can be active on the processor

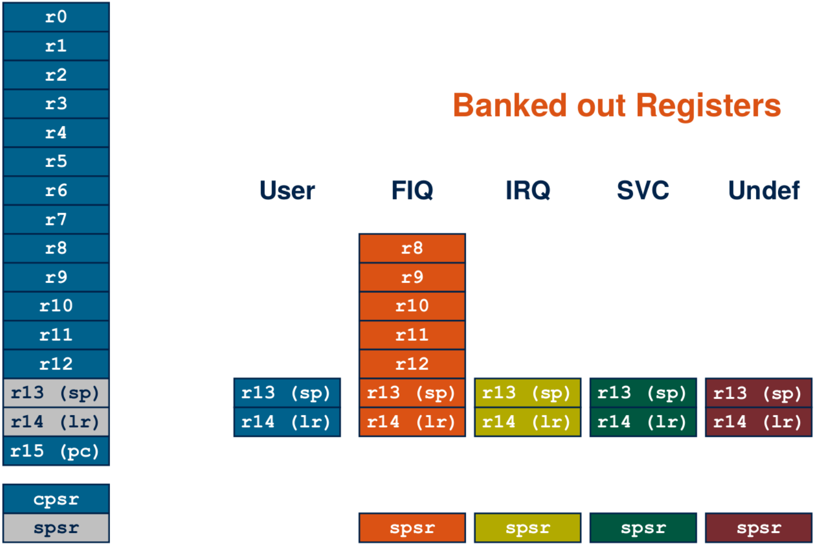
Processor modes

ARM has seven basic operating modes:

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

Registers

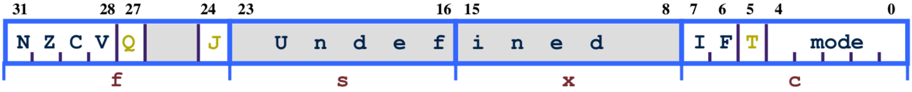
The current processor mode governs which of several banks is accessible



Register purpose

- **r0-r10** general purpose
- **r11 b**ase **p**ointer
  - holds the pointer to the current stack frame
- **r12 i**ntra **p**rocedure call scratch register
  - used by a subroutine to store temporary data
- **r13 s**tack **p**ointer
  - holds the pointer to the top of the stack
- **r14 l**ink **r**egister
  - holds the return addresses whenever a subroutine is called with a branch and link instruction
- **r15 p**rogram **c**ounter
  - holds the address of the next instruction to be executed
- **cpsr c**urrent **p**rogram **s**tatus **r**egister
  - holds processor status and control information
- **spsr s**aved **p**rogram **s**tatus **r**egister
  - accessed only in privileged modes

Program Status Registers



Condition code flags

- N = **N**egative result from ALU
- Z = **Z**ero result from ALU
- C = ALU operation **C**arried out
- V = ALU operation o**V**erflowed

Mode bits

- Specify the processor mode

Interrupt Disable bits

- I = 1: Disables the IRQ
- F = 1: Disables the FIQ

Carry vs Overflow

Carry is used for unsigned arithmetic, overflow if sign bit is corrupted

T Bit - architecture xT only

T=0 ⇒ processor in ARM state
T=1 ⇒ processor in Thumb state

Outline

1. Background
2. Advanced RISC Machine architecture
3. ARM assembly language
   - Arithmetic instructions
   - Condition codes
   - Data transfer instructions
   - Control transfer instructions
   - Stack instructions
4. ARM32 Architecture Procedure Calling Standard

First example

Let's start our introduction using a simple example in C:

```c
int total = 0;
for (int i = 10; i > 0; --i) {
        total += i;
}
```

It can be translated into those instructions supported by ARM's ISA:

```
        MOV  R0, #0          ; R0 accumulates total
        MOV  R1, #10         ; R1 counts from 10 down to 1
again   ADD  R0, R0, R1      ; R0 <- R0 + R1
        SUBS R1, R1, #1      ; R1 <- R1 - 1 , if R1==0 then Z <- 1
                             ; memento: Z == 1 means Zero result from ALU
        BNE  again           ; if Z != 1 branch to 'again' label
halt    B    halt            ; infinite loop to stop computation
```

Registers (repetita iuvant)

- R0 and R1 are references to registers, which are places in a processor for storing data during computation.
- Each register stores a single x-bit number ($x = 16, 32, 64$)
- Though registers store data, they are very separate from the notion of memory (it typically exists outside of the processor)
- Accessing memory takes more time than accessing registers (10x)

Instruction

Each line is an instruction that consists of two parts:

1. the **opcode** such as *MOV* that is an abbreviation indicating the type of operation
2. after the opcode comes **arguments** such as *R0*, *#0*

N° of arguments

Each opcode has strict requirements on the allowed arguments.

For example, a MOV instruction must have exactly 2 arguments:

1. must identify a register
2. must provide either a register or an immediate (prefixed by a #)

Immediate

A constant placed directly in an instruction is called an **immediate**, since it is immediately available to the processor when reading the instruction

Outline

1 Background

2 Advanced RISC Machine architecture

3 ARM assembly language
- Arithmetic instructions
- Condition codes
- Data transfer instructions
- Control transfer instructions
- Stack instructions

4 ARM32 Architecture Procedure Calling Standard

Shift and rotate instructions

Shift/rotate using a value contained in a register:

```
<op> Rd, Rs
```

Shift/rotate using an immediate value:

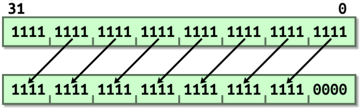
```
<op> Rd, Rm, #expr
```

`<op>` can be:

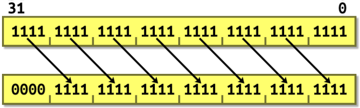
- LSL – Logical Shift Left - multiplication by 2^n $\Leftrightarrow$ << in C
- LSR – Logical Shift Right - unsigned division by 2^n $\Leftrightarrow$ >> in C
- ASR – Arithmetic Shift Right - signed division by 2^n
- ROR – Rotate Right - logical rotate by *n* bits

Shift and rotate operations - examples

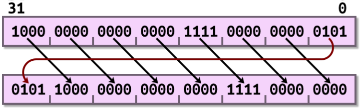
Logical Shift Left by 4



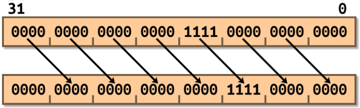
Logical Shift Right by 4



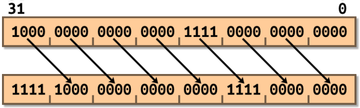
Rotate Right by 4



Arithmetic Shift Right by 4 with a positive value



Arithmetic Shift Right by 4 with a negative value

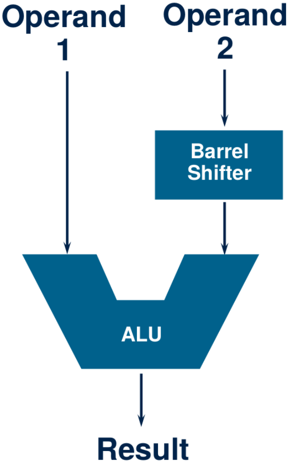


Data processing operations

```
<operation>{<cond>}{S} Rd, Rn, Operand2
```

- These instructions only work on registers, NOT on memory
- `<operation>`
  - Arithmetic: `ADD`, `ADC`, `SUB`, `SBC`, `RSB`, `RSC`
  - Logical: `AND`, `ORR`, `EOR`, `BIC`
  - Comparisons: `CMP`, `CMN`, `TST`, `TEQ`
    - set flags only - they do not specify Rd
  - Data movement: `MOV`, `MVN`
    - data movement does not specify Rn
- `{<cond>}` instructions can execute conditionally w.r.t. CPSR's flags
- `{S}` the CPSR register is updated based on the result of the arithmetic operation (e.g. *SUB**S***),
- `Operand2` is sent to the ALU via barrel shifter

The Second Operand



It can take one of 3 different forms:

1. Immediate value
   - `MOV r0, #42`
   - `ORR r1, r1, #0xFF00`
2. Registers shifted by values
   - `MOV r2, r2, LSR #1`
   - `RSB r10, r5, r14, ASR #14`
3. Registers shifted by registers
   - `BIC r11, r11, r1, LSL r0`
   - `CMP r9, r8, ROR r0`

Second Operand with shift/rotate operations - examples

- `MOV r0, r0, LSL #1`
  - Multiply R0 by two
- `MOV r1, r1, LSR #2`
  - Divide R1 by four (unsigned)
- `MOV r2, r2, ASR #2`
  - Divide R2 by four (signed)
- `MOV r3, r3, ROR #16`
  - Swap the top and bottom halves of R3
- `ADD r4, r4, r4, LSL #4`
  - Multiply R4 by 17
  - $N * 17 = N * (16 + 1) = N * 16 + N$
- `RSB r5, r5, r5, LSL #5`
  - Multiply R5 by 31
  - $N * 31 = N * (32 - 1) = N * 32 - N$

Summary of instructions so far

```
AND Rd, Ra, argb    ;   Rd <- Ra & argb
EOR Rd, Ra, argb    ;   Rd <- Ra ^ argb
SUB Rd, Ra, argb    ;   Rd <- Ra - argb
RSB Rd, Ra, argb    ;   Rd <- argb - Ra
ADD Rd, Ra, argb    ;   Rd <- Ra + argb
ADC Rd, Ra, argb    ;   Rd <- Ra + argb + carry
SBC Rd, Ra, argb    ;   Rd <- Ra - argb - !carry
RSC Rd, Ra, argb    ;   Rd <- argb - Ra - !carry
TST Ra, argb        ;   set flags for Ra & argb
TEQ Ra, argb        ;   set flags for Ra ^ argb
CMP Ra, argb        ;   set flags for Ra - argb
CMN Ra, argb        ;   set flags for Ra + argb
ORR Rd, Ra, argb    ;   Rd <- Ra | argb
BIC Rd, Ra, argb    ;   Rd <- Ra & ~argb
MOV Rd, arg         ;   Rd <- arg
MVN Rd, arg         ;   Rd <- ~argb
```

Multiply Instructions 32×32 to 32

```
MUL{S}{cond} Rd, Rn, Rm
```

- multiplies the values from `Rn` and `Rm`
- places the least significant 32 bits of the result in `Rd`

```
MLA{S}{cond} Rd, Rn, Rm, Ra
```

- multiplies the values from `Rn` and `Rm`
- adds the value from `Ra`
- places the least significant 32 bits of the result in `Rd`

```
MLS{cond} Rd, Rn, Rm, Ra
```

- multiplies the values from `Rn` and `Rm`
- subtracts the result from the value from `Ra`
- places the least significant 32 bits of the final result in `Rd`

Multiply Instructions 32×32 to 64 (1)

```
Op{S}{cond} RdLo, RdHi, Rn, Rm
```

UMULL

- interprets the values from Rn andRm as unsigned integers
- it multiplies these integers
- places the least significant 32 bits of the result in RdLo
- places the most significant 32 bits of the result in RdHi

UMLAL

- interprets the values from Rn and Rm as unsigned integers
- it multiplies these integers
- adds the 64-bit result to the 64-bit unsigned integer contained in RdHi and RdLo

Multiply Instructions 32×32 to 64 (2)

```
Op{S}{cond} RdLo, RdHi, Rn, Rm
```

SMULL

- interprets the values from Rn and Rm as 2-complement signed int
- it multiplies these integers
- places the least significant 32 bits of the result in RdLo
- places the most significant 32 bits of the result in RdHi

SMLAL

- interprets the values from Rn and Rm as 2-complement signed int
- it multiplies these integers
- adds the 64-bit result to the 64-bit signed integer contained in RdHi and RdLo

Outline

1. Background
2. Advanced RISC Machine architecture
3. ARM assembly language
   - Arithmetic instructions
   - Condition codes
   - Data transfer instructions
   - Control transfer instructions
   - Stack instructions
4. ARM32 Architecture Procedure Calling Standard

Condition codes

Each ARM instruction may incorporate a **condition code**:

- it specifies hat the operation should take place only when certain combinations of the flags hold
- it can be specified by including it as part of the opcode
- it usually comes at the end of the opcode
- it precedes the optional S on the basic arithmetic instructions
- based on the supposition that the flags were set based on a CMP or some *S instruction

Example

B**NE** the branch only takes place if the Z flag is 0
ADD**EQ**S perform an addition if the Z flag is 1

Condition codes list

Code	Suffix	Description	Flags
0000	EQ	Equal / equals zero	Z
0001	NE	Not equal	!Z
0010	CS / HS	Carry set / unsigned higher or same	C
0011	CC / LO	Carry clear / unsigned lower	!C
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	!N
0110	VS	Overflow	V
0111	VC	No overflow	!V
1000	HI	Unsigned higher	C and !Z
1001	LS	Unsigned lower or same	!C or Z
1010	GE	Signed greater than or equal	N == V
1011	LT	Signed less than	N != V
1100	GT	Signed greater than	!Z and (N == V)
1101	LE	Signed less than or equal	Z or (N != V)
1110	AL	Always (default)	any

Condition codes examples

- Use a sequence of several conditional instructions

```
if (a==0) func(1);
```

```
CMP r0,#0
MOVEQ r0,#1
BLEQ func
```

- Set the flags, then use various condition codes

```
if (a==0) x=0;
if (a>0) x=1;
```

```
CMP r0,#0
MOVEQ r1,#0
MOVGT r1,#1
```

- Use conditional compare instructions

```
if (a==4 || a==10) x=0;
```

```
CMP r0,#4
CMPNE r0,#10
MOVEQ r1,#0
```

Example: Hailstone sequence

```c
int i=0, n=5;
while (n != 1) {
      i++;
      if (n % 2 == 1) n = 3*n + 1;
      else n = n/2;
}
```

```
.text
.global _start
_start: MOV  R0, #5         @ R0 == n == 5 is current number
        MOV  R1, #0         @ R1 == i == 0 is the counter of iterations
        B again
again:  ADD  R1, R1, #1     @ increment number of iterations
        ANDS R0, R0, #1     @ test whether R0 is odd
        BEQ  even
        ADD  R0, R0, R0, LSL #1 @ if odd, set R0 = R0 + (R0 << 1)
        ADD  R0, R0, #1     @ R0 = R0 + 1
        B    again          @ and repeat (guaranteed R0 > 1)
even:   MOV  R0, R0, ASR #1 @ if even, set R0 = R0 >> 1
        SUBS R7, R0, #1     @ and repeat if R0 != 1
        BNE  again
```

Task 1 - Euclid's GCD algorithm

```
x = 40;
y = 25;
while (x != y) {
     if (x > y) x -= y;
     else       y -= x;
}
```

Implement in ARM assembly the Euclid's GCD algorithm.
Start with $R0 = 40$ and $R1 = 25$.

Task 1 - Euclid's GCD algorithm

```
x = 40;
y = 25;
while (x != y) {
     if (x > y) x -= y;
     else       y -= x;
}
```

Implement in ARM assembly the Euclid's GCD algorithm.
Start with $R0 = 40$ and $R1 = 25$.

```
        MOV R0, #40       @ R0 is x
        MOV R1, #25       @ R1 is y
again   CMP R0, R1
        SUBGT R0, R0, R1  @ GT signed greater than
        SUBLT R1, R1, R0  @ LT signed less than
        BNE again         @ NE not equal
halt    B halt
```

Task 2

```c
#include <stdio.h>

extern int mystery(int); /* mystery assembler routine */
int main() {
        static const char str[] = "Hello, World!";
        const int len = sizeof(str) / sizeof(str[0]);
        char newstr[len];
        for (int i = 0; i < len; i++) newstr[i] = mystery(str[i]);
        printf("%s\n", newstr);
        return 0;
}
```

```
mystery @ r0 == str[i]
        SUB     r1, r0, #0x41
        CMP     r1, #0x5a - 0x41
        ADDLS   r0, r0, #0x61 - 0x41
        MOV     pc, r14
        END
```

Task 2 - solution

```
int mystery(int c)
{
        unsigned int t;
        t = c - 'A';
        if (t <= 'Z' - 'A') c += 'a' - 'A';
        return c;
}
```

The tricky thing here is the coercion to unsigned int which allows us to replace two comparisons with a single one.
We can write it in a more expected way like this:

```
int mystery2(int c)
{
        if (c >= 'A' && c <= 'Z') c += 'a' - 'A';
        return c;
}
```

Task 3

Write your own ARM assembly language routine to compute the factorial of an input number N.

$$n! = \prod_{k=1}^{n} k = n \cdot (n-1) \cdots 3 \cdot 2 \cdot 1$$

- Don't worry about the $N == 0$ case: just get the basics working.
- Don't try disassembling the C version until you've first had a go!
- On entry, N is stored in R0. Use R1 for the loop counter.

Examples

$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$
$7! = 7 \cdot 6 \cdot 5! = 42 \cdot 120 = 5040$

Task 3 - solution

```
fact:                           @ On entry, N is stored in R0
        MOVS    r1, r0          @ R1 is our loop counter. Copy N to R1 and test
        MOVEQ   r0, #1          @ if (R1 == 0) Set result to 1 and fall through end
loop:   SUBNES  r1, r1, #1      @ if (R1 != 0) Decrement R1 and test
        MULNE   r0, r1, r0      @ if (R1 != 0) Result = R1 * Result
        BNE     loop            @ if (R1 != 0) Loop
        MOV     pc, r14         @ Return with result in R0 (see AAPCS later)
```

Outline

1. Background
2. Advanced RISC Machine architecture
3. ARM assembly language
   - Arithmetic instructions
   - Condition codes
   - Data transfer instructions
   - Control transfer instructions
   - Stack instructions
4. ARM32 Architecture Procedure Calling Standard

Basic memory instructions

```
<op>{size}{cond} Rd, <address>
```

The ARM supports memory access via two instructions:

1. **LDR** instruction loads data out of memory
   - Rd $\leftarrow$ value at $\langle$address$\rangle$
2. **STR** stores data into memory
   - value at $\langle$address$\rangle$ $\leftarrow$ Rd

{size} can be any one of:

- **B** unsigned Byte (Zero extend to 32 bits on loads.)
- **SB** Signed Byte (LDR only. Sign extend to 32 bits.)
- **H** unsigned Halfword (Zero extend to 32 bits on loads.)
- **SH** Signed Halfword (LDR only. Sign extend to 32 bits.)
- - omitted, for Word.

Basic memory - example

An assembly program fragment that adds the integers in an array

- R0 holds the address of the first integer of the array
- R1 holds the number of integers in the array
- R4 holds the sum of the of integers in the array

```
addInts:    MOV  R4, #0
addLoop:    LDR  R2, [R0]    @ indexed addressing mode
            ADD  R4, R4, R2
            ADD  R0, R0, #4
            SUBS R1, R1, #1
            BNE  addLoop
```

Addressing modes

Name	ARM example
Register direct	`MOV R0, R1`
Direct	`LDR R0, 0x<address>`
Immediate	`MOV R0, #15`
Indexed, base	`LDR R0, [R1]`
Pre-indexed, base with displacement	`LDR R0, [R1, #4]`
Pre-indexed, autoindexing	`LDR R0, [R1, #4]!`
Post-indexing, autoindexed	`LDR R0, [R1], #4`
Double Reg indirect	`LDR R0, [R1, R2]`
Double Reg indirect, with scaling	`LDR R0, [R1, R2, LSL #2]`
Program Counter relative	`LDR R0, [PC, #offset]`

Addressing Mode	Assembly Mnemonic	Address	R1=
Pre-indexed, displacement	`LDR R0, [R1, #d]`	R1 + d	R1
Pre-indexed, autoindexing	`LDR R0, [R1, #d]!`	R1 + d	R1 + d
Post-indexing, autoindexed	`LDR R0, [R1], #d`	R1	R1 + d

PC-relative addressing

Addresses can be represented as a PC-relative expression

- it is represented in the instruction as the PC value plus or minus a numeric offset: `[pc, #offset]`
- the code is position-independent, i.e. it can be loaded anywhere in memory without the need to adjust any addresses
- the value of the PC is the address of the current instruction plus
  - 8 bytes in ARM state
  - 4 bytes in Thumb state (with subtle details)

Example:

```
10078: e59f1010  ldr   r1, [pc, #16]   ; pc == 10080
1007c: e3a0200e  mov   r2, #14
10080: e3a07004  mov   r7, #4
...
10090: 00020094  .word 0x00020094      ; 10090-10080=10 == #16
```

PC relative addressing Hello World example

```
.data
string: .asciz "Hello World!\n"
len = . - string

.text
.global _start

_start: mov r0, #1
        ldr r1, =string
        ldr r2, =len
        mov r7, #4
        svc 0
_exit:
        mov r7, #1
        svc 0
```

```
# as -o hello.o hello.s
# ld -o hello hello.o
```

```
# objdump -d example
example: file format elf32-littlearm

Disassembly of section .text:

00010074 <_start>:
10074: e3a00001  mov   r0, #1
10078: e59f1010  ldr   r1, [pc, #16]
1007c: e3a0200e  mov   r2, #14
10080: e3a07004  mov   r7, #4
10084: ef000000  svc   0x00000000

00010088 <_exit>:
10088: e3a07001  mov   r7, #1
1008c: ef000000  svc   0x00000000
10090: 00020094  .word 0x00020094
```

Multiple-register memory operations

```
<op><mode>{cond} Rn{!}, <reglist>
```

ARM includes instructions allowing several values to be loaded or stored in the same instruction:

- `LDM`: `<reglist>` := values@Rn
- `STM`: values@Rn := `<reglist>`

`<mode>` controls *how Rn is incremented*:

- `<op>`IA – Increment After (default)
- `<op>`IB – Increment Before
- `<op>`DA – Decrement After
- `<op>`DB – Decrement Before

`<reglist>` is the list of registers to load or store

- it can be a comma-separated list or an Rx-Ry style range.

Multiple-register memory operations - example 1

```
LDMIA R0!, { R5-R8 }
```

1. the ARM processor looks into the R0 register for an address
2. it loads into R5 the four bytes starting at that address
3. into R6 the four bytes starting from $R0 + 0x4$
4. into R7 the four bytes starting from $R0 + 0x8$
5. into R8 the four bytes starting from $R0 + 0xC$

R0!

The effect of exclamation mark is to write back the final address into R0, so at the end of this operation R0 is stepped forward by $0x10 = 16_{10}$

Multiple-register memory operations - example 2, 3

- `LDMIA R0!, {R3, R7}`
    1. Load words addressed by R0 into R3 and R7
    2. Increment After each load
    3. Write back the final address into R0

- `LDMIA R0, { R1-R4, R8, R11-R12 }`
    1. will load seven words from memory
    2. Increment After each load
    3. the order in which the registers are listed is not significant!
        - `{ R1-R4, R8, R11-R12 } == { R11-R12, R8, R1-R4 }`
        - R1 will receive the first word loaded from memory

Multiple-register memory operations - example 4

Shift every number in an array into the next spot:

- R0 holds address of first integer in array
- R1 holds array's length
- this code works only if array's length is multiple of 4

```
      MOV R4, #0
loop: LDMIA R0, { R5-R8 }  @ R0 isn't modified
      STMIA R0!, { R4-R7 } @ R0 modified for next iteration
      MOV R4, R8
      SUBS R1, R1, #4
      BNE loop
```

E.g.: the array $< 1, 2, 3, 4, 5, 6, 7, 8 >$ becomes $< 0, 1, 2, 3, 4, 5, 6, 7 >$

Outline

1 Background

2 Advanced RISC Machine architecture

3 ARM assembly language
- Arithmetic instructions
- Condition codes
- Data transfer instructions
- Control transfer instructions
- Stack instructions

4 ARM32 Architecture Procedure Calling Standard

Branch operations

`<op>{X}{cond} <address>`

Branch instructions are used to alter control flow

- **B** - Branch
  - R15|PC ← `<address>`
- **BL** - Branch with Link
  1. R14|LR ← `<address> of next instruction`
  2. R15|PC ← `<address>`

eXchange

`<op>X` instructions can change the processor state from ARM to Thumb, or from Thumb to ARM

Outline

1. Background
2. Advanced RISC Machine architecture
3. ARM assembly language
   - Arithmetic instructions
   - Condition codes
   - Data transfer instructions
   - Control transfer instructions
   - Stack instructions
4. ARM32 Architecture Procedure Calling Standard

The Stack

The stack is a data structure that stores information about the active subroutines of a computer program

- it keep track of the point to which each active subroutine should return control when it finishes executing
- an **active subroutine** is one that has been called but is yet to complete execution after which control should be handed back to the point of call
- such activations of subroutines may be nested to any level (recursive as a special case), hence the stack structure

R13 the Stack Pointer

It is always 4 byte aligned and contains the memory location's address occupied by the top of the stack

Operations for the stack

STM and LDM provide a mechanism for storing state on the stack:

- **FD = Full Descending**
  - **STMFD/LDMFD ⇔ STMDB/LDMIA**
- ED = Empty Descending
  - STMED/LDMED ⇔ STMDA/LDMIB
- FA = Full Ascending
  - STMFA/LDMFA ⇔ STMIB/LDMDA
- EA = Empty Ascending
  - STMEA/LDMEA ⇔ STMIA/LDMDB

Full Descending

Anything but a full descending stack is rare ⇒ PUSH and POP instruction

PUSH

`PUSH{cond} reglist`

Stores registers on the stack, with the lowest numbered register using the lowest memory address and the highest numbered register using the highest memory address

- uses `[SP, #-4]` as the highest memory address
- updates SP to point to the location of the lowest stored value
- PUSH $\Leftrightarrow$ STMFD | STMDB

POP

`POP{cond} reglist`

Loads registers from the stack, with the lowest numbered register using the lowest memory address and the highest numbered register using the highest memory address.

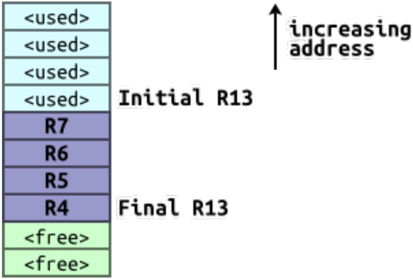
- uses the value in the SP register as the lowest memory address (FD)
- updates the SP register to point to the location immediately above the highest location loaded
- POP $\Leftrightarrow$ LDMFD | LDMIA

POP {..., PC}

If a POP instruction includes PC|R13 in its reglist, a **branch** to this location is performed when the POP instruction has completed

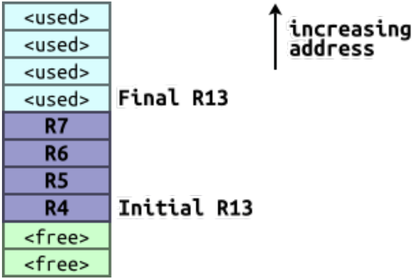
Operations for the stack - example

STMFD r13!, {r4-r7}



PUSH {r4-r7}

LDMFD r13!, {r4-r7}



POP {r4-r7}

Outline

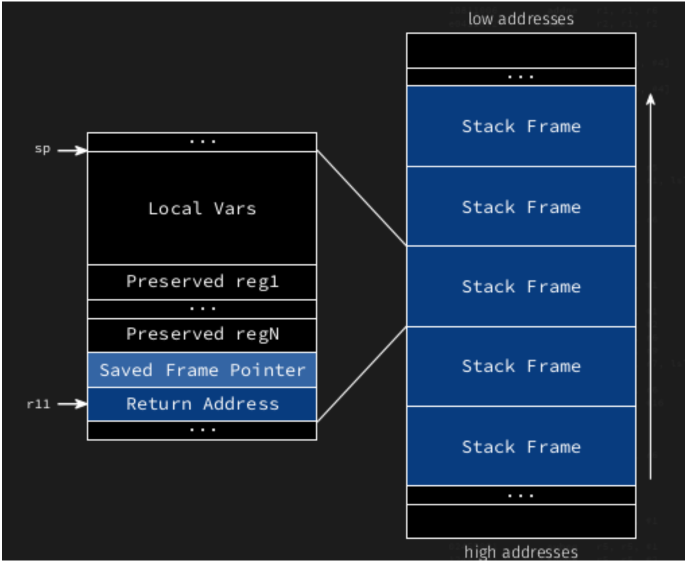
1. Background
2. Advanced RISC Machine architecture
3. ARM assembly language
   - Arithmetic instructions
   - Condition codes
   - Data transfer instructions
   - Control transfer instructions
   - Stack instructions
4. ARM32 Architecture Procedure Calling Standard

ARM Architecture Procedure Calling Standard

The ARM Architecture Procedure Calling Standard (**AAPCS**) governs how procedures call each other in high-level languages such as C:

- **R0-R3** are used as parameters and **R0** as return value
  - any subsequent parameters are passed on the stack
  - *caller-saved* if needed
  - hold temporary quantities that need not be preserved across calls
- **R4-R11** are *callee-saved* registers
  - hold long-lived values that should be preserved across calls
- **R12** (aka IP) is a scratch register
  - useful for veneers and other linkage tricks (PLT, GOT, etc)
- **R13-R15** (aka SP, LR, PC) are special registers

Stack Frames



Stack Frames

- functions take advantage of Stack for saving local variables, preserving register state, etc.
- to keep everything organized, functions use Stack Frames
  - localized memory portion within the stack
  - each frame is dedicated for a specific function
- a stack frame gets created in the prologue of a function
- R11/FP is set to the bottom of the stack frame
- the stack frame (starting from it's bottom) generally contains:
  - return address (i.e., the previous LR)
  - previous Frame Pointer
  - any registers that need to be preserved
  - function parameters (in case the function accepts more than 4)
  - local variables
- a Stack Frame gets destroyed during the epilogue of a function

Function structure

The structural parts of a function are 3:

1. prologue:
   - save the previous state of the program (by storing values of LR and R11 onto the Stack)
   - set up the Stack for the local variables of the function
2. body:
   - is usually responsible for some kind of unique and specific task
   - sets the result in R0
3. epilogue:
   - restore the program's state to it's initial one (before the function call) so that it can continue from where it left of
   - readjust the Stack Pointer using the Frame Pointer register (R11) as a reference and performing add or sub operation
   - restore the previously saved register values by poping them from the Stack into respective registers

Function structure

```
.global main

nonleaf: @ A prologue of a non-leaf function
        push    {fp, lr}     @ Start of the prologue. Saving FP and LR onto the stack
        add     fp, sp, #0   @ Setting up the bottom of the stack frame
        sub     sp, sp, #16  @ End of the prologue. Allocating some buffer on the stack
        @ BODY
        @ An epilogue of a non-leaf function
        sub     sp, fp, #0   @ Start of the epilogue. Readjusting the Stack Pointer
        pop     {fp, pc}     @ End of the epilogue. Restoring Frame pointer from the stack,
                             @ jumping to previously saved LR via direct load into PC


leaf:
        @ A prologue of a leaf function
        push    {fp}         @ Start of the prologue. Saving FP onto the stack
        add     fp, sp, #0   @ Setting up the bottom of the stack frame
        sub     sp, sp, #12  @ End of the prologue. Allocating some buffer on the stack
        @ BODY
        @ An epilogue of a leaf function
        add     sp, fp, #0   @ Start of the epilogue. Readjusting the Stack Pointer
        pop     {fp}         @ restoring frame pointer
        bx      lr           @ End of the epilogue. Jumping back to main via LR register
```

The End

Yes... it's over!