Topic 2:
Microprocessors and Memory Hierarchy in Microprogrammed Embedded Systems

Systèmes Embarqués Microprogrammés

Prof. David Atienza Alonso, SEL-STI

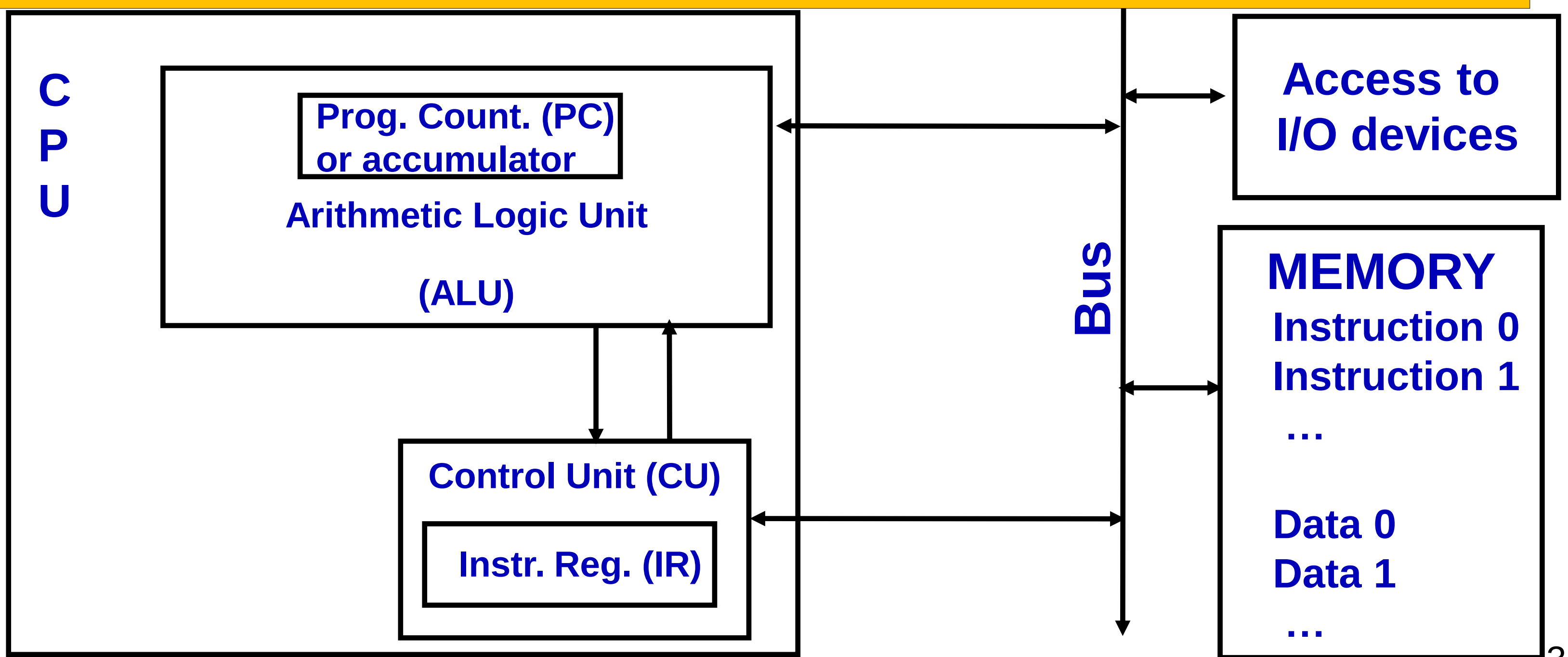
Content of Session

- Introduction to microprocessors design and functionality
  - Block diagram and types of instruction sets
  - Link between memory, registers and CPU
  - Microprocessor metrics: performance and power, why ARM?
- ARM microprocessor architectures for the NDS
  - NDS history and boot-up for processor configuration
  - Memory access sizes and addressing modes
  - Register file design and microprocessor modes
- ARM instruction set
  - Memory access (read/writing) instructions
  - Arithmetic-logic instructions
  - Jump instructions
- Assembly programming for ARM processor
  - Conditional and loop construction
  - How to include assembly in C programs
  - Passing parameters between C and assembly: Registers and stack

Microprocessor architectures in computing systems

- **Central Processing Unit (CPU)**: the primary element performing the computing function, it is the unit that carries out each instruction of a program stored in memory in sequence, to perform the basic arithmetical, logical, and input/output operations of the system
- **Microprocessor:** programmable machine that incorporates on a single integrated circuit (IC, or microchip) all the functions of a CPU

“Stored program architecture“ or Von Neumann Architecture [1945]



Microprocessor architectures in computing systems

- **Central Processing Unit (CPU)**: the primary element performing the computing function, it is the unit that carries out each instruction of a program stored in memory in sequence, to perform the basic arithmetical, logical, and input/output operations of the system
- **Microprocessor:** programmable machine that incorporates on a single integrated circuit (

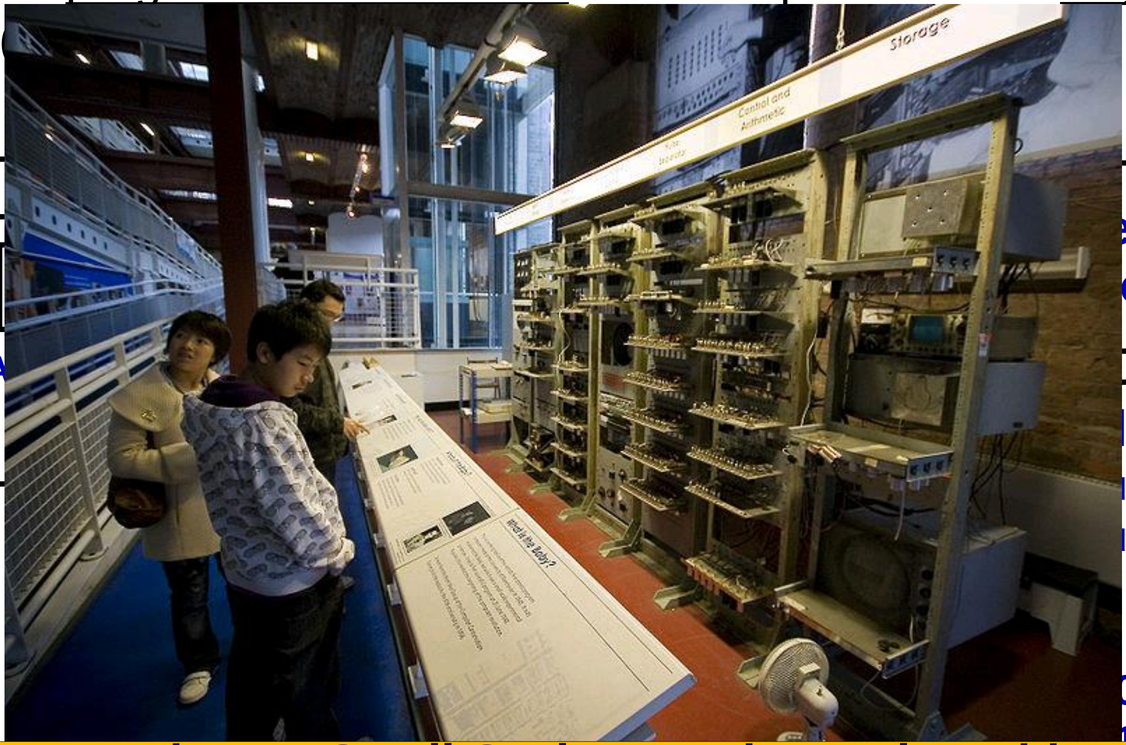
C
P
U

A

ess to
evices

ORY
ction 0
ction 1

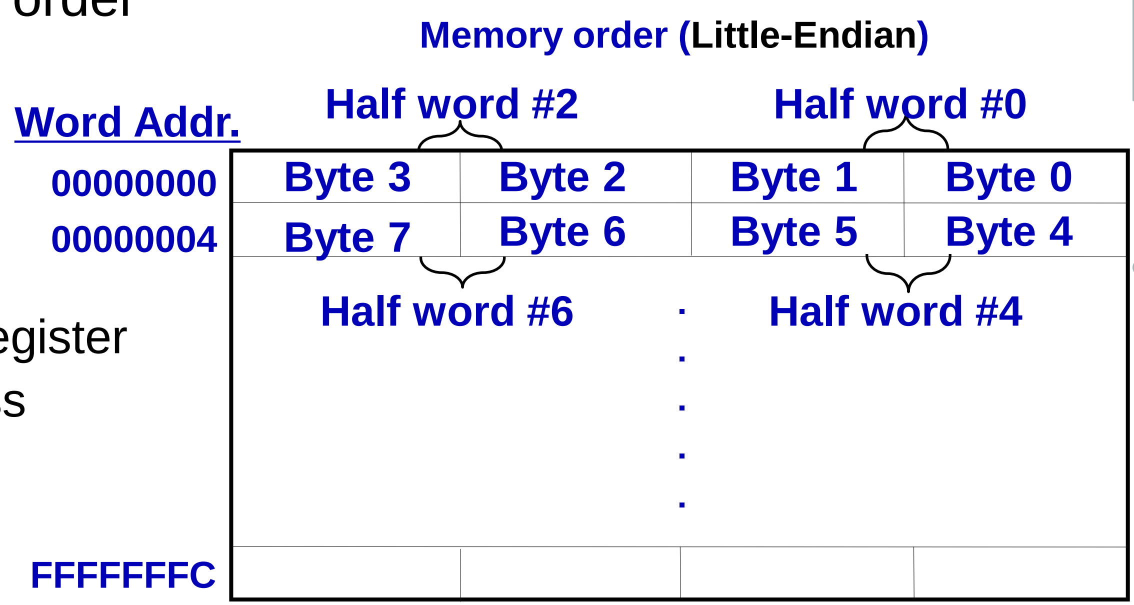
0
1



Manchester Small-Scale Experimental Machine (SSEM), nicknamed “Baby” [June 21st, 1948]

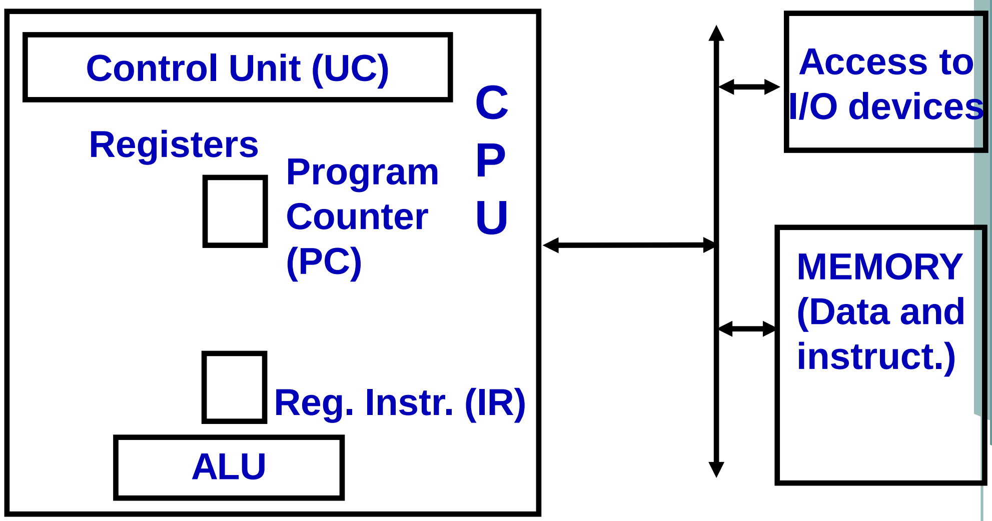
Main features of Von Neumann architecture

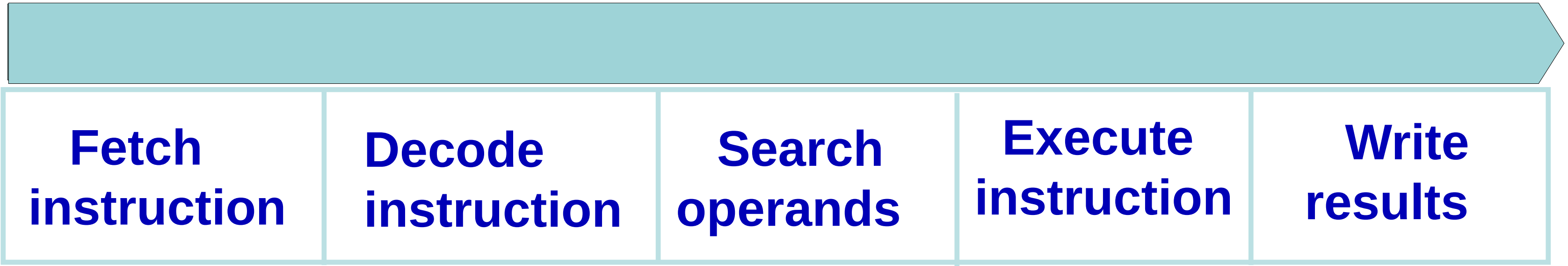
- It is based on the concept of a **memory stored program**
  - The memory content is organized in **words**, all of the same size (e.g., 32 bits)
  - Words in memory follow a linear order (i.e., addresses), with endianness
- Two main memory contents, but no explicit distinction between them
  - **Instructions**: program that controls what the microprocessor has to do
  - **Data**: pieces of information that the program processes and generates
- The instructions execution order is **sequential** (0, 1, …)
  - Determined by their addresses order in the memory
- The Program Counter (PC) register keeps next instruction address to execute



Main features of Von Neumann architecture

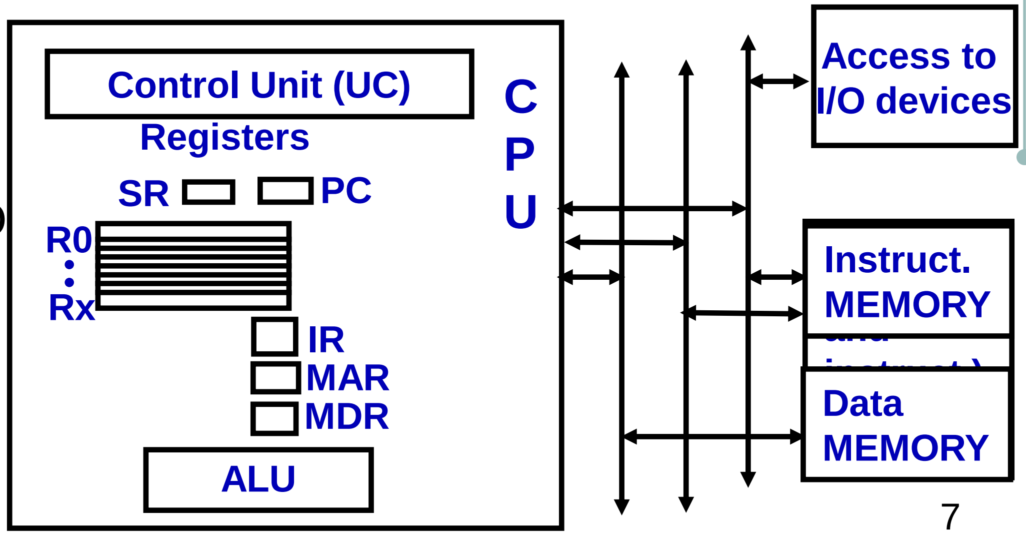
- Five phases in program instruction execution
  1. Search instruction in memory *(Fetch)* and calculation next instruct. address
  2. Decoding instruction by the CPU
  3. Search of instruction operands (in memory or registers)
  4. Execution of the instruction (in ALU if an arithmetic one)
  5. Write results of operation (in memory or registers)





Variations of baseline Von Neumann architecture

- Two alternatives in the definition of the instruction set of the CPU
  - **Complex instruction set computer (CISC)**: design strategy <u>to bridge the semantic gap</u> through single instructions that execute several low-level operations (e.g., load from memory, arithmetic operat., and memory store).
    - Examples: Intel x86, Motorola 68x, VAX, System/360, or PDP-11
  - **Reduced instruction set computer (RISC)**: design strategy based on <u>simple instructions</u> to provide higher performance through <u>much faster execution of each instruction</u>, i.e., **load-store architectures**
    - Examples: ARM, MIPS, Atmel AVR, Power, SPARC or AMD 29k.
- Harvard architecture
  - Architectures with <u>physically separated</u> storage and signal buses for instruct., data and I/O
    - Examples: ARM , MIPS, Blackfin, PowerPC



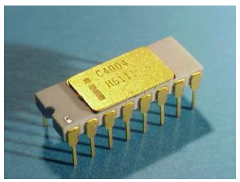
Trying to fix two problems:
(1) memory wall and
(2) interconnects saturation

Microprocessor architectures: Evolution to microprogrammed embedded systems

- Microprocessor have evolved since Nov. 15th, 1971 towards more complex functionality and performance, but at a cost of power…

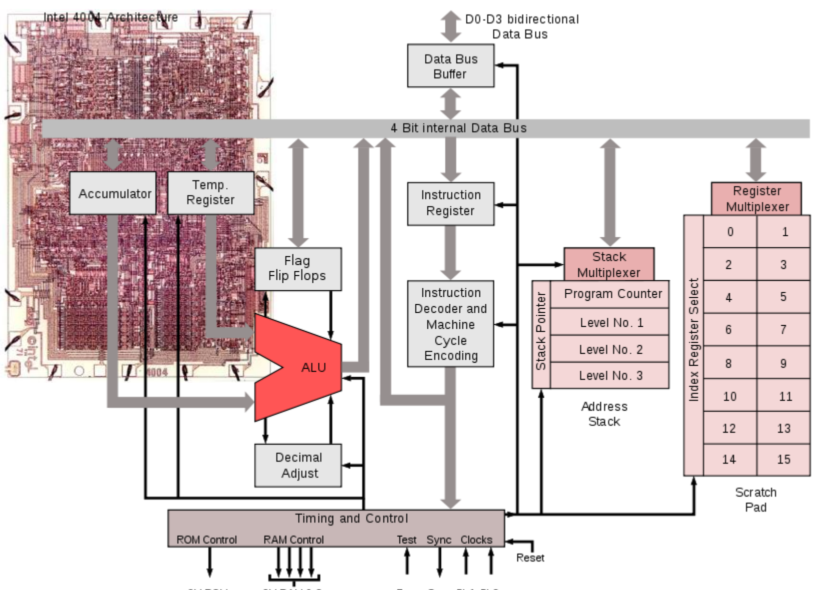
0.3 Watts **95-130 Watts!**

\- +

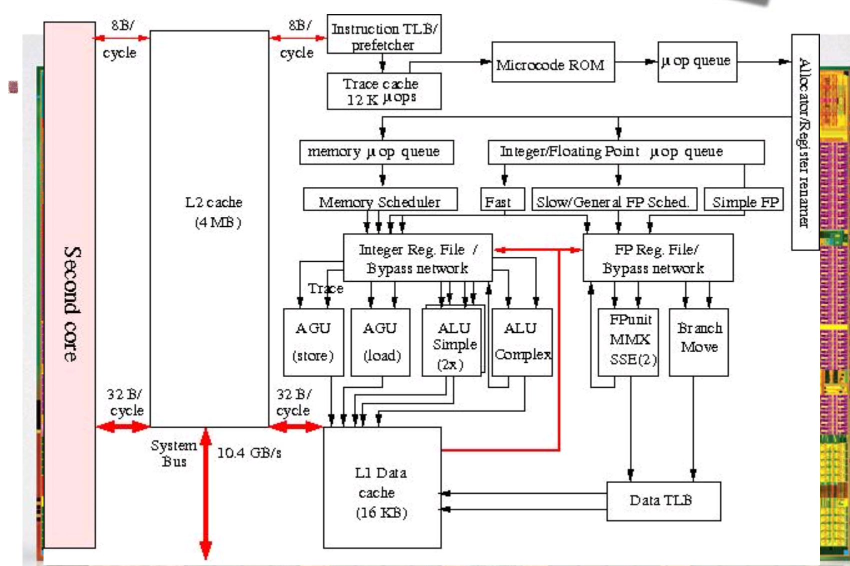


Intel 4004 [1971]
(2250 p-MOS trans., 108 KHz, 4-bit,mem.:640B)

Intel Xeon [2006-]
(820M trans., 3.73GHz, 64-bit, mem.:4MB/L2)



Busicon Calculator 141-PF



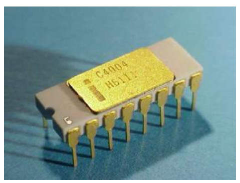
Intel Many Integrated Core (MIC) [2011]

Microprocessor architectures: Evolution to microprogrammed embedded systems

- Microprocessor have evolved since Nov. 15th, 1971 towards more complex functionality and performance, but at a cost of power…

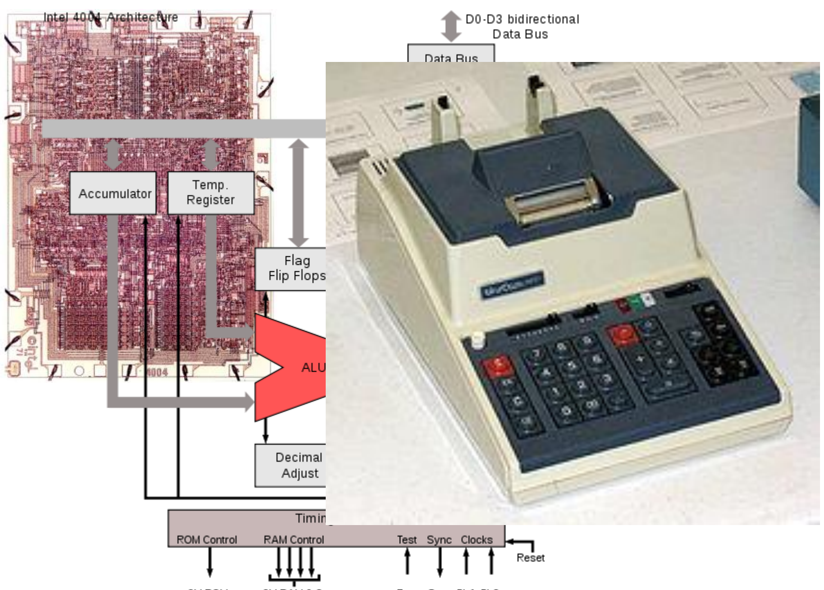
0.3 Watts **95-130 Watts!**

\- +

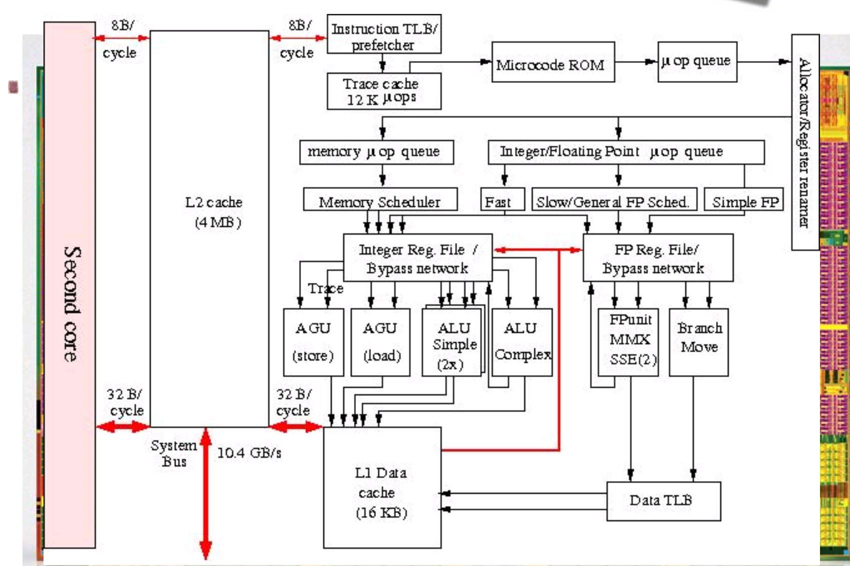


Intel 4004 [1971]
(2250 p-MOS trans., 108 KHz, 4-bit,mem.:640B)

Intel Xeon [2006-]
(820M trans., 3.73GHz, 64-bit, mem.:4MB/L2)



Busicon Calculator 141-PF



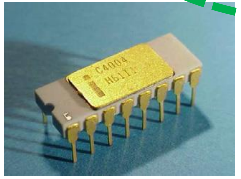
Intel Many Integrated Core (MIC) [2011]

Microprocessor architectures: Evolution to microprogrammed embedded systems

- Microprocessor have evolved since Nov. 15th, 1971 towards more complex functionality and performance, but at a cost of power…

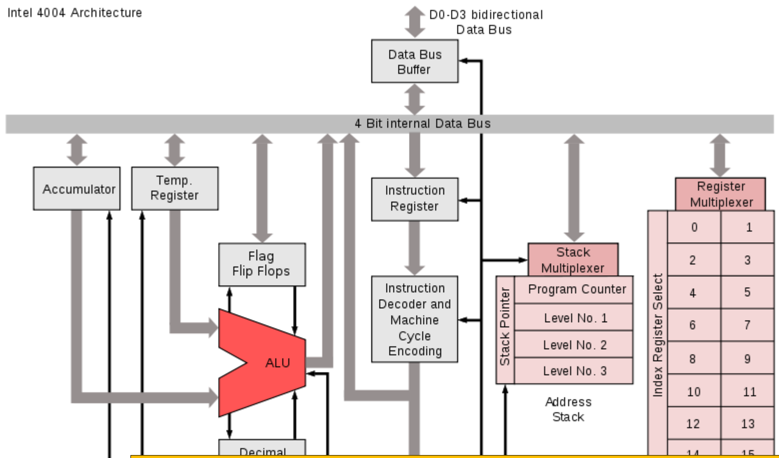
0.3 Watts — ARM — 95-130 Watts!

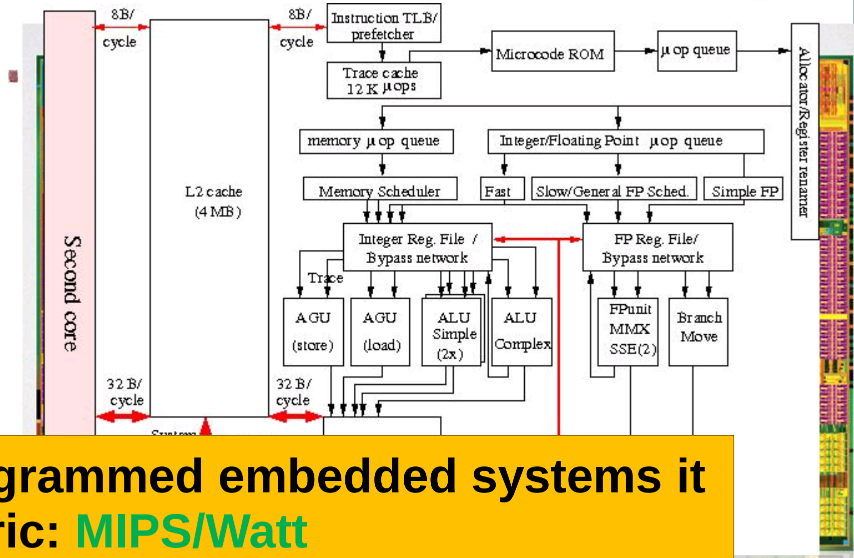
\- +



Intel 4004 [1971]
(2250 p-MOS trans., 108 KHz, 4-bit,mem.:640B)

Intel Xeon [2006-]
(820M trans., 3.73GHz, 64-bit, mem.:4MB/L2)





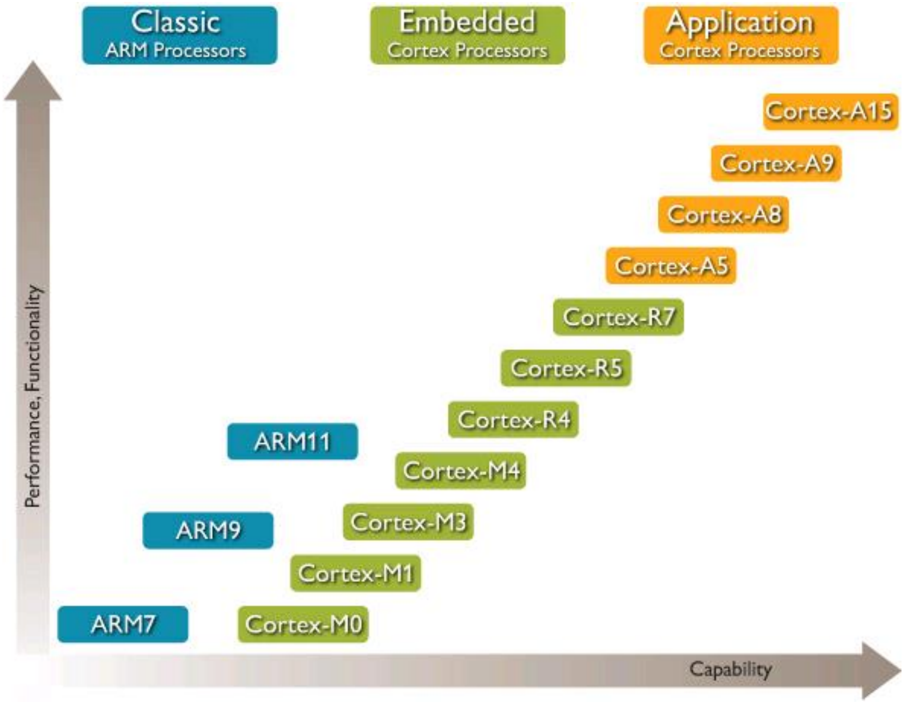
Principles remain, but in microprogrammed embedded systems it is key a new metric: MIPS/Watt
New opportunities for business!

2011]

Advanced RISC Machines (ARM) for all flavours of microprogrammed embedded systems

- Advanced RISC Machines Ltd (ARM) was officially founded in 1990
- **Reduced Instruction Set Computers (RISC)** for energy-efficient microprogrammed computing
  - Instructions with minimum semantic (move data, add/sub/mult. 2 numbers, ...)
  - ARM2 [1986], probably simplest 32-bit core ever: 30K transistors, no cache
  - Large range of ARM designs: <u>always basic RISC architecture + extensions</u>

- Business model:
  - Licensing ARM designs to semiconductor partners who fabricate and sell to customers “à la carte”



Advanced RISC Machines (ARM) for all flavours of microprogrammed embedded systems

- Advanced RISC Machines Ltd (ARM) was officially founded in 1990
- **Reduced Instruction Set Computers (RISC)** for energy-efficient microprogrammed computing
  - Instructions with minimum semantic (move data, add/sub/mult. 2 numbers, ...)
  - ARM2 [1986], probably simplest 32-bit core ever: 30K transistors, no cache
  - Large range of ARM designs: <u>always basic RISC architecture + extensions</u>

- Business model:
  - Licensing ARM designs to semiconductor partners who fabricate and sell to customers “à la carte”
  - ARM does not fabricate silicon itself, **but 90% of all microprogrammed embedded systems** use one or more ARM processors!

By Jun. 2015, **16.9 Billion** ARM cores shipped,
By Feb. 2021, **180 Billion** ARM cores shipped
Best sellers: **ARM9 and ARM7TDMI**

Families of the ARM processor

- All of them are compatible with their core RISC architecture:
  - 32-bit instruction set architecture, 16 x 32-bit reg. file, load/store architecture

Family	Microprocessor	Version	Examples of microprog. embedded systems
ARM1	ARM1	v1	Evaluation system BBC
ARM2	ARM2, ARM250	v2	Acorn Archimedes, Chessmachine
ARM3	ARM2a	v2a	Idem
ARM6	ARM60, ARM600, ARM610	v3	Acorn Risc PC 600, Apple Newton 100 series
ARM7	ARM700, ARM710	v3	Apple eMate 300, Psion Series 5
ARM7TDMI	**ARM7TDMI**, ARM710T, ARM720T, ARM740T	v4T	Game Boy Advance, **Nintendo DS**, Apple iPod
StrongARM	SA-110, SA-1110	v4	Apple Newton 2x00 series, Ipaq H36x0, Zaurus SL-5x00
ARM9TDMI	ARM9TDMI, ARM920T, ARM940T	v4T	Hewlet Packard HP-49/50 Calculators, Sun SPOT
ARM9E	**ARM946E-S**, ARM926EJ-S	v5TE	**Nintendo DS**, Nokia N-Gage, Nokia 32xx, Sony Ericsson (series K/W)
XScale	PXA210/PXA250, PXA255, PXA27x, PXA900	v5TE	iPAQ H3900, HTC Universal, Dell Axim, Motorola Q, Blackberry Pearl
ARM11	ARM1136J(F)-S, ARM1176JZ(F)-S	v6	Zune, Nokia N93, Apple iPhone, Motorola Z8

The history of the Nintendo DS (NDS) processors

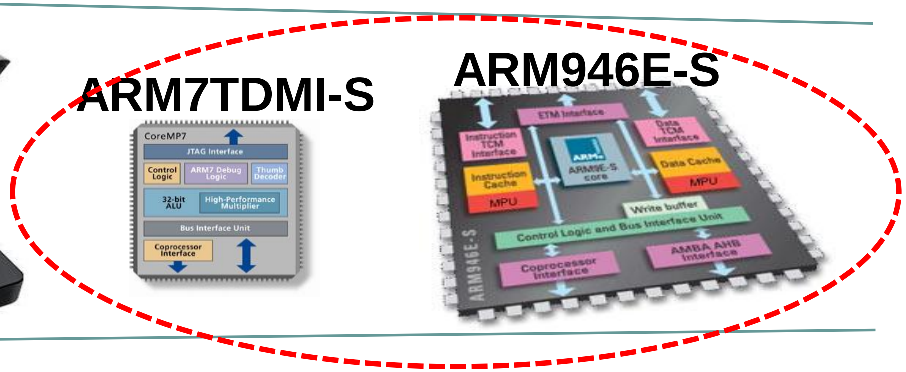
- Precursor: Game Boy Advance (GBA), Mar. 2001
  - Screen: 2.9 inches, TFT LCD (240x160 pixels)
  - Processors
    - ARM 7TDMI, 16.78 MHz: user sw processing
    - Z80, 8MHz: microcontroller for I/O support

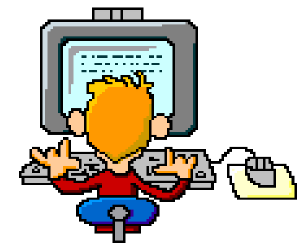
- Nintendo DS, Nov. 2004; and DS Lite, Mar. 2006
  - Screen: 3 vs 3.1 inches, TFT LCD (256 x 192 pixels)
  - Processors
    - ARM 946E-S, 67.028 MHz: user software processing and main I/O peripherals
    - ARM 7TDMI-S,16.78 MHz: dedicated I/O support (sound, wifi and specific keys)

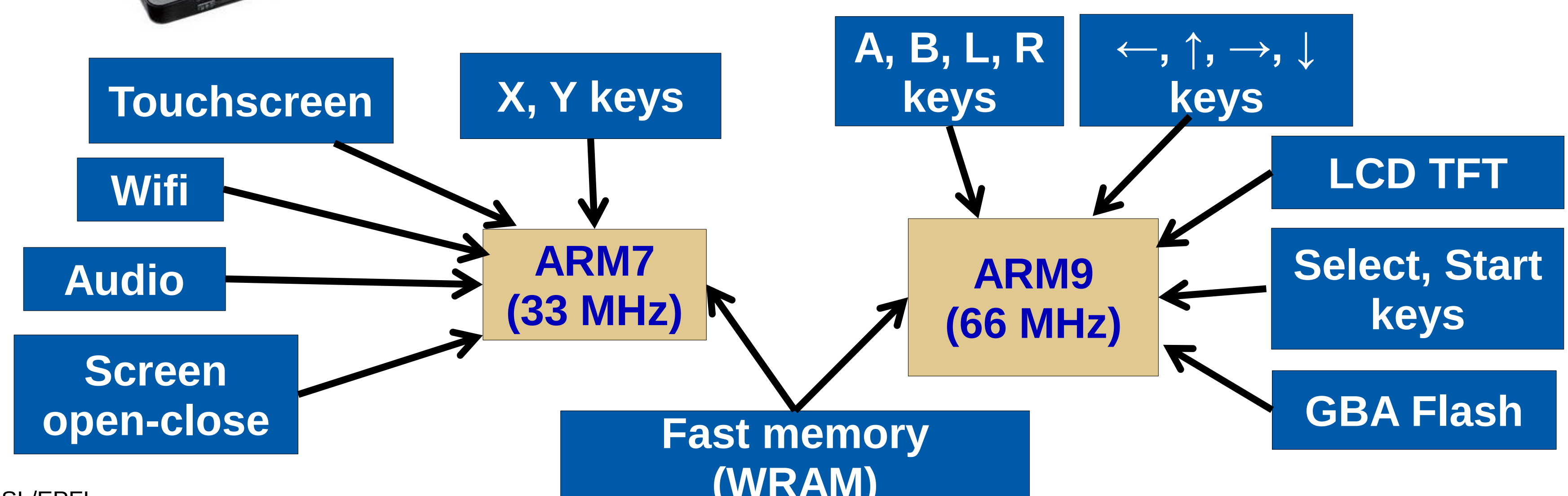
- Nintendo DSi, Nov. 2008, and DSi XL, Oct. 2009
  - Screen: 3.2 vs. 4.2 inches, TFT LCD (256 x 192 pixels)
  - Processors
    - ARM 946E-S, 133 MHz: user software processing and main I/O peripherals
    - ARM 7TDMI-S, 16.78 MHz: dedicated I/O support (sound, wifi and specific keys)

NDS processing and I/O management

- Two asymmetric 32-bit Advanced RISC Microprocessors (ARM) cores
  - ARM 946E-S: user software processing and main I/O peripherals
  - ARM 7TDMI-S: dedicated I/O support (sound, wifi and specific keys)







Nintendo DS memory range and start-up to initialize ARM processors

- Cartridge ROM: **0x08000000 - 0x09ffffff** (32MB)
- Main RAM: **0x02000000 - 0x023fffff** (4MB)
- BIOS (Basic Input/Output System) in ROM
  - ARM9: **0xffff0000 - 0xffff7fff** (32KB);
  - ARM7: **0x00000000 - 0x00003fff** (16KB)

Reset vector 0xffff0000

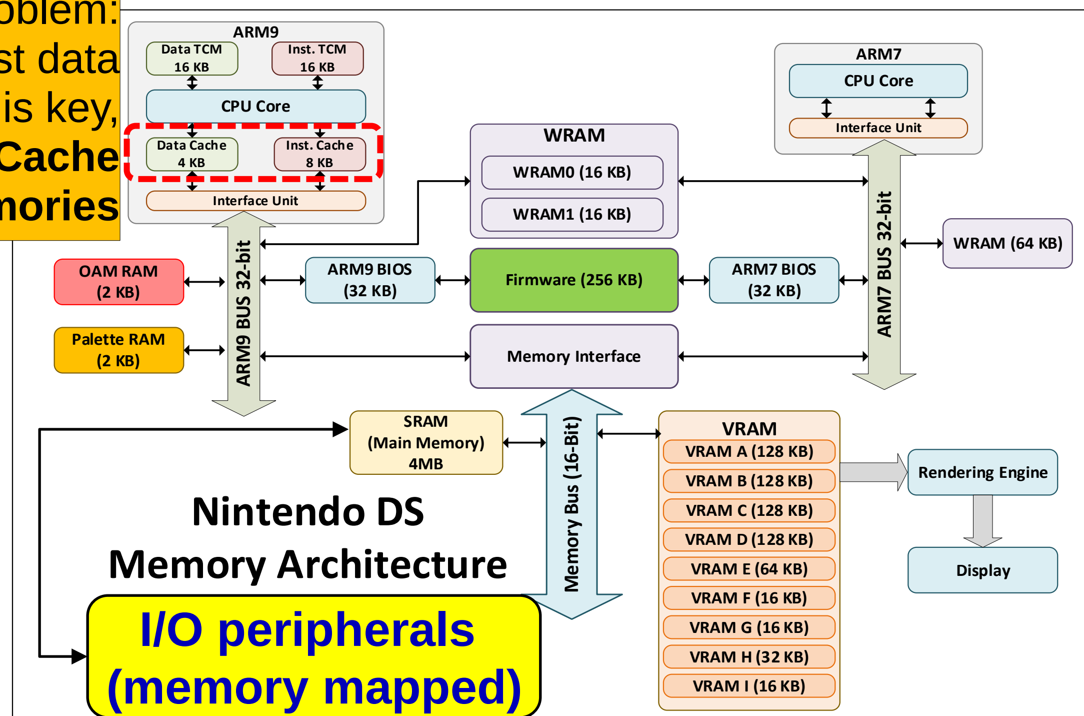
Reset vector 0x00000000

Word Addr.	Memory order (Little-Endian)			
00000000	Byte 3	Byte 2	Byte 1	Byte 0
00000004	Byte 7	Byte 6	Byte 5	Byte 4
	Half word		.	
			.	
			.	
			.	
			.	
(1GB) FFFFFFFC				

Solving mem. wall problem: Fast data access is key, with **Cache memories**

- Boot-up process configures the data instruction memories

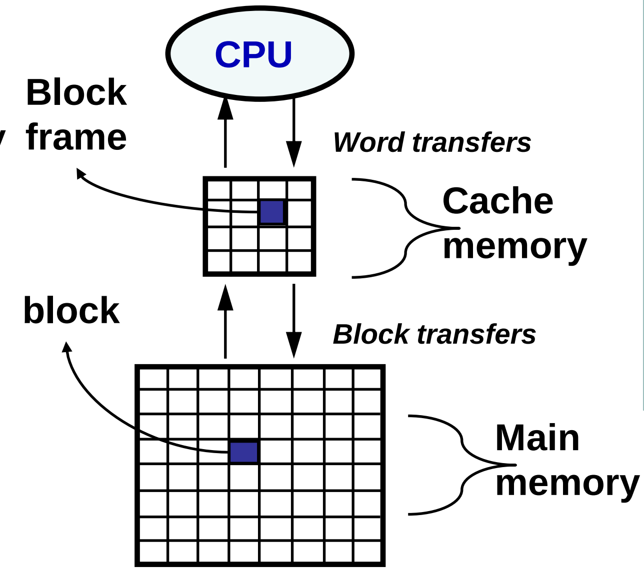
  1. Firmware in FLASH memory: menu, pictochat, and user preferences
  2. Decoding carried out from the BIOS into RAM
  3. Firmware copies the cartridge ROM to the RAM
  4. ARM cores start running

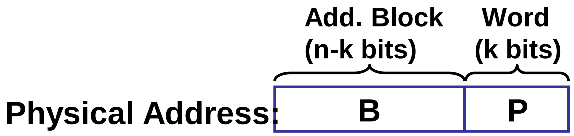


Memory hierarchy optimization: Caches

- Small and fast memory located between processor and main memory
  - Stores copy of memory portion currently in use
- **Objetive**: reduction of memory access time

- New memory structure: cache and main memory
  - *MM (Main memory):*
    - Composed of 2^n addressable words
    - "divided" in nB blocks of fixed size (2^K words per block)
    - Physical address
  - *CM (Cache memory) – Hardware controlled:*
    - nM block frames of 2^K words each (nM<<nB)
  - *SPM (Scratch-Pad memory) – Software controlled*
  - *Directory (in cache memory):*
    - Indicates which subset of nB blocks are located in nM block frames

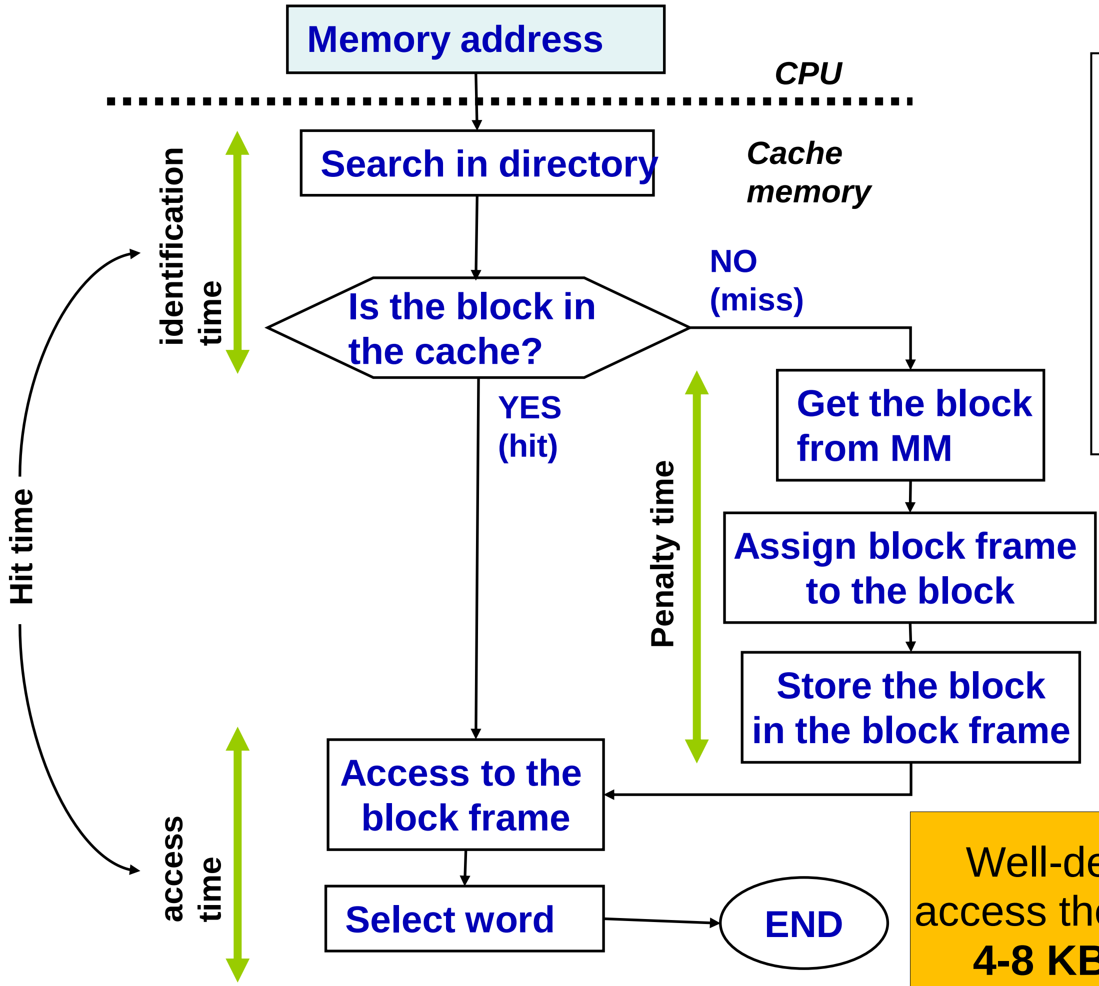




nB: number of blocks
nF: number of block frames
B: block address
F: address of block frame
P: word inside the block

Interaction cache-main memory

- Technological limitations require a complex interaction



- Maximize the percentage of hits
- Minimize access time
- Minimize the total penalty (misses)
- Reduce hardware cost

$$T_{total} = T_{hit} + (1 - H)T_{penalty}$$

(% hits ratio)

Well-designed programs have good locality: access the same data items over and over again:
4-8 KB caches in ARM = 80-90% hit ratio

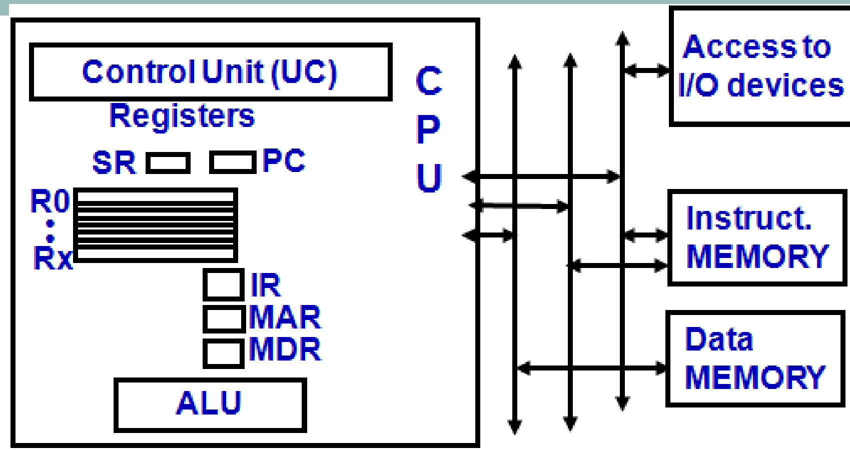
Main design aspects of cache memories

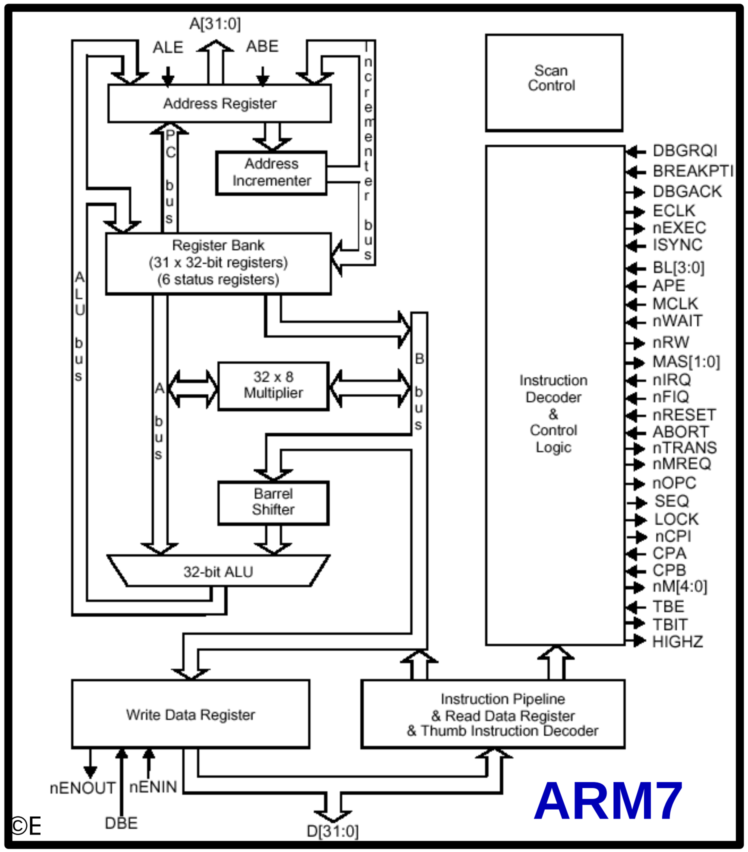
- Cache configuration:
  - Size
  - Block size
  - Cache levels
  - Unified or separated

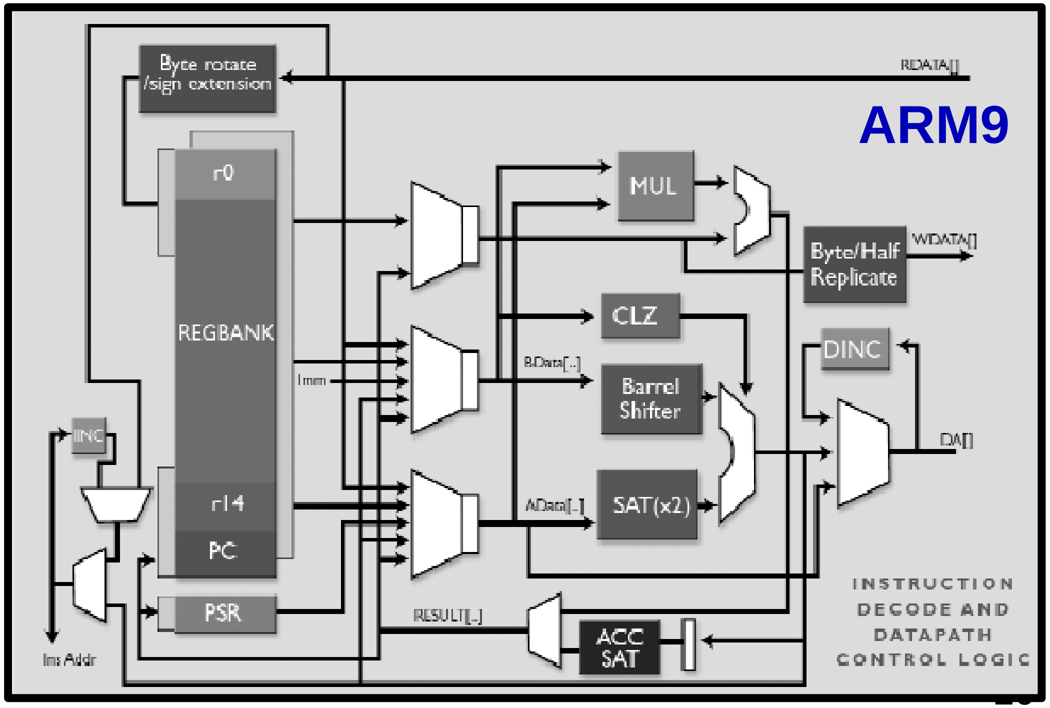
- Behavior policies
  1. **Placement policy**:
     - Needed as there are less block frames in CM than blocks in MM
     - Selects in which block frame each block can be loaded
  2. **Replacement policy**:
     - Needed as any new block in CM must replace one of the existing ones
     - Select which block to replace
  3. **Write policy**:
     - Needed to keep coherence between CM and MM
     - Selects when to update a block from MM after being modified in CM
  4. **Search policy**:
     - Determines which blocks (and when) should be loaded in CM

ARM processor architectures

- 32-bit RISC processor (32-bit instruction = word)
- 37 registers of 32-bits (16 available)
- With ALU, Multiplier, Shiffer and (often) caches
- Von Neuman-type (ARM7) and Harvard (ARM9)
- 8- / 16- / 32-bit data accesses

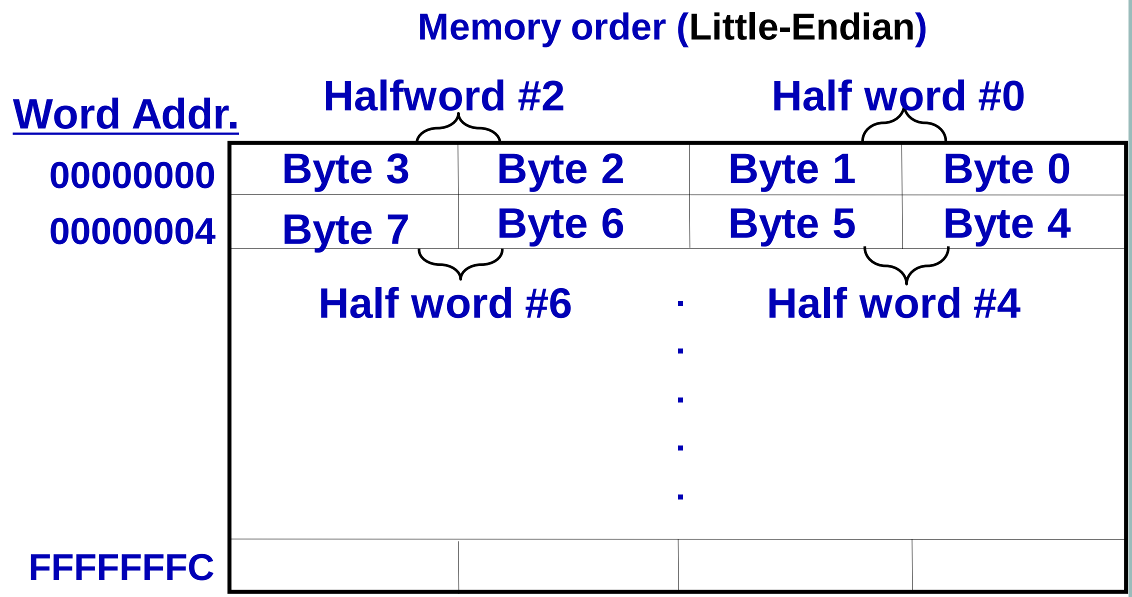






Allowed data sizes and instruction sets

- Possible data sizes in ARM:
  - **Word** means 32 bits (four bytes)
  - **Halfword** means 16 bits (two bytes)
  - **Byte** means 8 bits (one byte)
- By default: little-endian (big-endian is configurable)

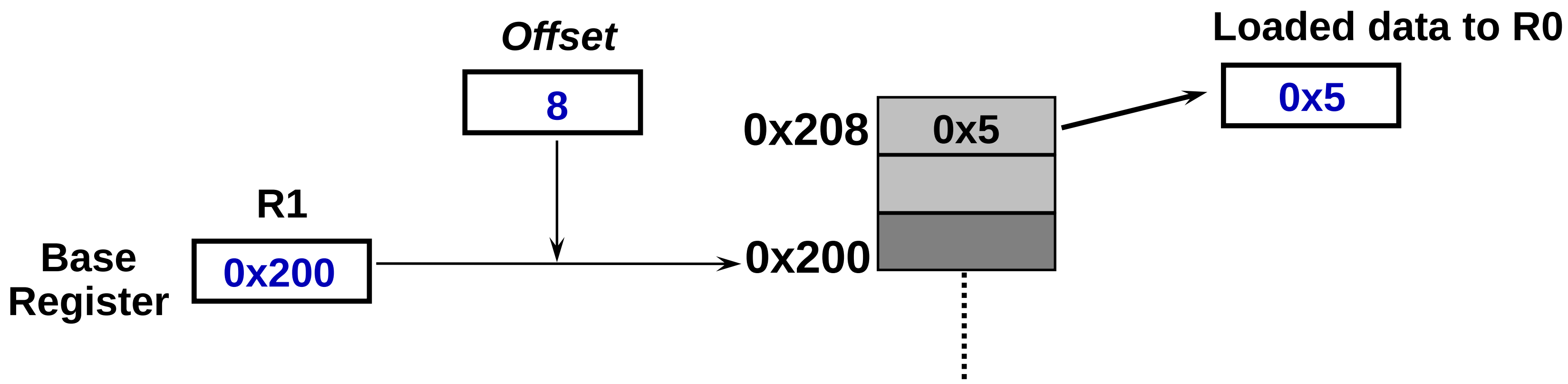


- Most ARM processors implement two instruction sets
  - Regular instructions of 1 word each: 32-bit ARM Instruction Set
  - Compact instructions of halfword: 16-bit Thumb Instruction Set
- Well-known extensions of the ARM instruction set
  - Jazelle: direct execution of Java bytecode
  - NEON: acceleration for multimedia and signal processing

Registers and addressing mode

- ARM includes always 37 registers of 32-bit width each
  - 1 program counter (PC)
  - 1 state register (SR)
    - indicates if last operation result was zero (Z), negative (N), overflow (O), etc.
  - 5 registers to store the program state when we switch operation mode in the processor
  - 30 registers of general purpose

- Base register + offset/displacement (e.g., LDR r0,[r1,#8])



ARM processor modes and registers access

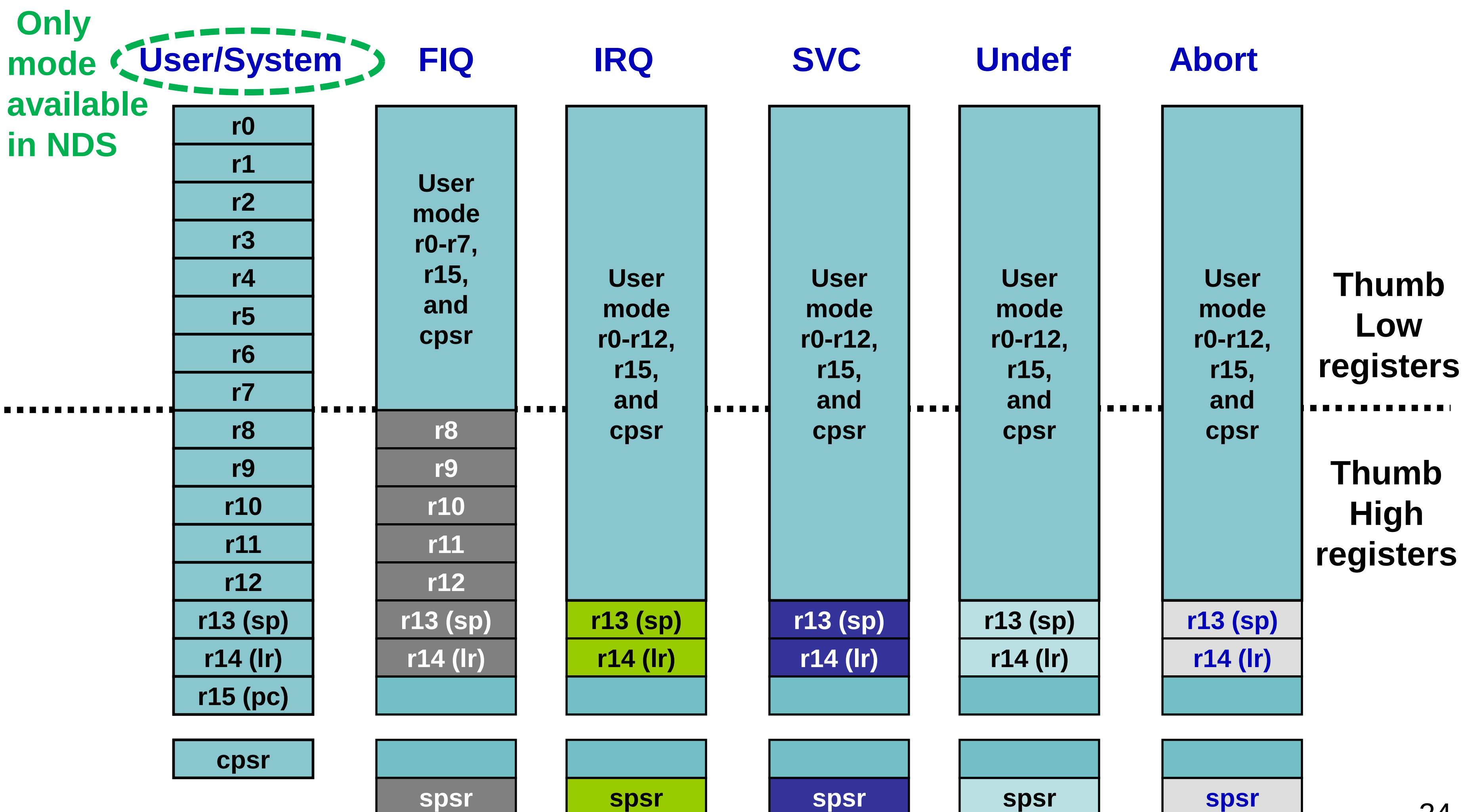
- The ARM architecture has seven basic operating modes, for different levels of priority and use of system resources
  - **User** : unprivileged mode under which most of the tasks should run
  - **FIQ** : when a high priority (fast) interrupt is raised
  - **IRQ** : when a low priority (normal) interrupt is raised
  - **Supervisor** : 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

Key protection mechanism for the microprogrammed embedded system to avoid geting blocked with malfunctioning program or I/O device!

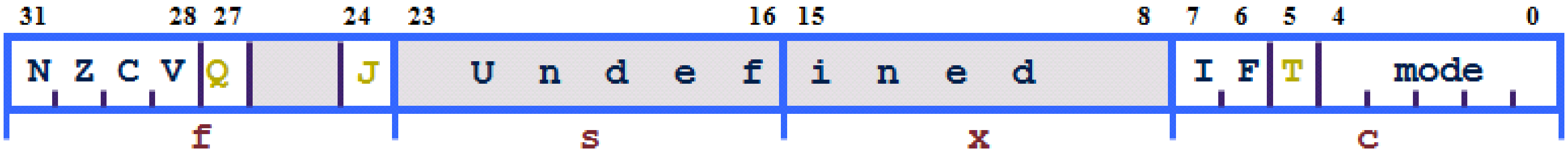
- Each processor mode selects a register file with 16 registers:
  - a particular set of ***r0-r12*** registers
  - a particular ***r13*** (the stack pointer, ***sp***) and ***r14*** (the link register, ***lr***)
  - the program counter, ***r15*** (***pc***)
  - the current program status register, ***cpsr***
- The privileged modes (except System) can also access:
  - a particular saved program status register, ***spsr***

Register organization summary

- The *System* mode uses the *User* mode register set



Program status registers (***cpsr***): Information about microprocessor status



Flags we will try in the NDS

- Condition code flags (or bits)
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation oVerflowed
- Interrupt Disable bits.
  - I = 1: Disables the IRQ.
  - F = 1: Disables the FIQ.
- Mode bits
  - Specify the processor mode

- **Sticky overflow flag - Q flag**
  - Architecture 5TE/J only (ARM9)
  - Indicates if saturation has occurred
- **T Bit**
  - Architecture xT only
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state
- **J bit**
  - Architecture 5TEJ only (ARM9)
  - J = 1: Processor in Jazelle state

ARM instruction set

- 5 types of assembly instructions
  - 3 fundamental RISC instructions types
    - Memory access instructions
    - Arithmetic-logic instructions
    - Control (jump) instructions
  - Two extended RISC instructions types for microprocessor control
    - Special instructions to manipulate the state register (***cpsr***)
    - Change of execution mode and type of instruction set (ARM/Thumb)

Memory access instructions

- Two basic operations
  - LDR: Load register content with value from memory address
  - STR: Store register content in memory address
- Syntax:
  - **LDR**{<cond>}{<size>} Rd, <address>
  - **STR**{<cond>}{<size>} Rd, <address>
    - Examples: **LDR r0,[r1,#8]**
      STR r0,[r1,#8]

- Memories in the system must support all sizes (default: Word size)
  - <size> = Signed (S) and then Byte (B), Halfword (H)

Examples:

LDR	**STR**	Word
LDRB	**STRB**	Byte
LDRH	**STRH**	Halfword
LDRSB		Load byte with sign
LDRSH		Load halfword with sign

Loading 32 bit constants into registers

- To allow larger constants to be loaded, the assembler offers a "pseudo-instruction":
  - **LDR rd, =const**

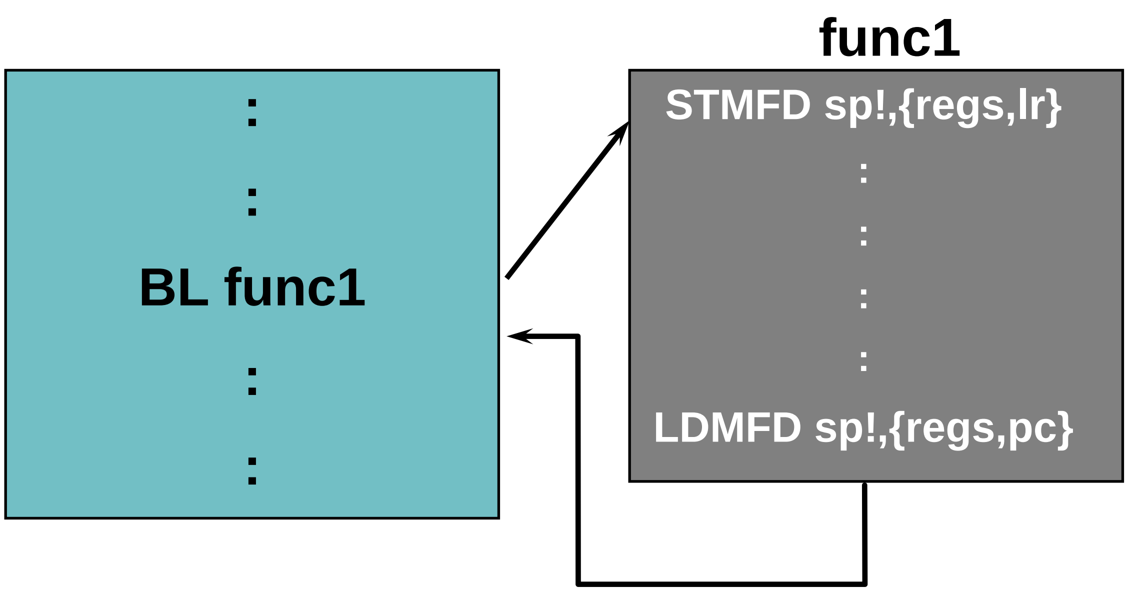
- For example
  - **`LDR r0,=0xFF`**
  - **`LDR r0,=0x55555555`**

Arithmetic-Logic instructions

- They only work between registers, types:
  - Arithmetic: **ADD ADC SUB MUL SBC RSB RSC**
  - Logic operations: **AND ORR EOR BIC**
  - Comparisons: **CMP CMN TST TEQ**
  - Data movements: **MOV MVN**
- Syntax: **<Operation>{<cond>}{S} Rd, Rn, Operand2**
  - Example: **ADD r0, r1, r2**
- Remarks
  - Comparisons only fix the flags (or bits) of ***cpsr*** (not the ***Rd*** value)
    - Example: **CMP r3,#0**
  - Data movements do not specify ***Rn***
    - Example: **MOV r0,#0xFF**

Jumps instructions: branches and subroutines

- **B <label>**
  - Jump that is ***PC*** relative, up to ±32MB range
- **BL <subroutine>**
  - Stores return address in ***LR*** (***r14***)
  - Returning implemented by restoring the ***PC*** from ***LR***



Conditional execution and flags in ARM instructions

- ARM instructions can be made to execute conditionally by post-fixing them using the condition code field: *{<cond>}*
  - This improves code density and performance by reducing the number of forward branch instructions

  - Example: **ADD{<cond>} Rd, Rn, Operand2**

```
  CMP    r3,#0
  BEQ    skip
  ADD    r0,r1,r2
skip:
```

```
CMP    r3,#0
ADDNE  r0,r1,r2
```

Accepted conditions in ARM instructions

- ***AL*** is defined by default and it is not needed to indicate it

Suffix	Description	Flags tested
EQ	Equal	**Z=1**
NE	Not equal	**Z=0**
CS/HS	Unsigned higher or same	**C=1**
CC/LO	Unsigned lower	**C=0**
MI	Minus	**N=1**
PL	Positive or Zero	**N=0**
VS	Overflow	**V=1**
VC	No overflow	**V=0**
HI	Unsigned higher	**C=1 & Z=0**
LS	Unsigned lower or same	**C=0 or Z=1**
GE	Greater or equal	**N=V**
LT	Less than	**N!=V**
GT	Greater than	**Z=0 & N=V**
LE	Less than or equal	**Z=1 or N=!V**
AL	Always	

Equivalence between C and ARM assembly: *if* constructions - general form

- Definition

```
if(cond) {instTrue;} else {instFalse;}
```

- Uses comparison (*CMP*) and conditional branch (*Bcond*) inst.

```
CMP ifCondition
B<NOT cond> caseFalse
    ; execution instruct. case true
    ....
    B regroup
  caseFalse:
    ; execution instructs. case false
   ....
  regroup:
    ; insts. with regrouped flow
```

- Example

```
if (a==7) {count+=1;}
else {count-=1;}
...
```

  - ***a*** is stored in r0 ; ***count*** is stored in r1

- Solution

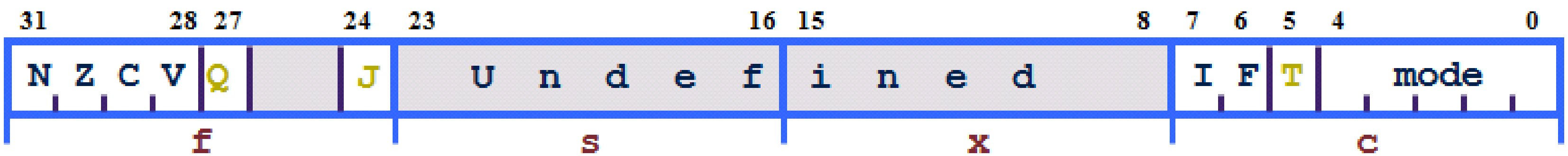
```
  CMP r0,#7
  BNE caseFalse
  ADDS r1,r1,#1
  B regroup
caseFalse:
  SUBS r1,r1,#1
regroup:
  ...
```

Equivalence between C and ARM assembly: compact ARM conditional constructions

- ARM assembly for an ***if-else*** construction

```
if (a==7) {count+=1;}
else {count-=1};
```

  - ***a*** is stored in **r0 ;** ***count*** is stored in **r1**



  - Condition code flags (or bits)

    N = Negative result from ALU ; Z = Zero result from ALU

    C = ALU operation Carried out ; V = ALU operation oVerflowed

- Solution

```
CMP     r0, #7          ; compare a with 7
ADDEQ   r1, r1, #1      ; do count=count++, if a == 7
SUBNE   r1, r1, #1      ; do count=count--, if a != 7
```

Equivalence between C and ARM assembly: *while* loops – general form

- Definition

```
while (cond) {instructsTrue;} after_loop_instructs
```

- Uses comparison (*CMP*) and conditional branch (*Bcond*) inst.

```
loop:
        CMP cond
        BNE <cond> caseFalse
        ; execute instruc. true condition
        B loop
 caseFalse:
        ; inst. after loop
```

- Example

```
while(a<3) {
    count+=1;
    a++;
}
…
```

  - ***a*** is stored in **r0** ; ***count*** is stored in **r1**

- Solution

```
loop:
   CMP r0,#3
   BGE caseFalse
   ADD r1,r1,#1
   ADD r0,r0,#1
   B loop
 caseFalse:
   …
```

Equivalence between C and ARM assembly: ***for*** loops can use general form of ***while*** loops

- Example:

```
 count=0;
for (i=10;i>0;i--) count+=1;
```

  - ***i*** is stored in **r0;** ***count*** is stored in **r2**

- Solution:

```
          MOV r2, #0
          MOV r0, #10
loop:
          CMP r0, #0
          BLE caseFalse
          ADD r2, r2, #1
          SUBS r0, r0, #1
          B loop
caseFalse:
          ; instructions after loop
```

Developing an assembly program

1. Declaration of variables and memory address aliases
   - Variables can be initialized
2. Define name (“main”) of functions as ***.global*** label
3. Start of the code section using: ***.text*** {label}
4. Define the end of the program: ***.end*** {label}

```
.equ x, 45
.equ y, 64
.equ stack_top, 0x1000

.global _start

.text
_start:
        MOV sp, #stack_top
        MOV r0, #x
        STR r0, [sp]
        MOV r0, #y
        LDR r1, [sp]
        ADD r0, r0, r1
        STR r0, [sp]
stop:
        B stop
.end
```

Including assembly in C programs: Two methods

- Directly in the C program
  - Using for each assembly line: ***asm("assembly Instruct.");***

```
int main() {
   ...
   asm("mov R1,R2");
   ...
```

- Use in C source files external assembly files
  - Declare in C the use of external functions: ***EXTERN Prototype_funct;***
  - Declare assembly functions as global symbols: ***.GLOBAL Name_funct***
  - Implement the function in the ***.text*** region of the program: ***Name_funct:***

```
EXTERN funct1(int);
int main() {
   int result, a=5;
   result=funct1(a);
   ...
}
```

```
.GLOBAL funct1
.text
   funct1:
         ...
         ...
```

How parameters are passed in ARM
Example: C program with a function

```
extern int delay1(int);
main()
{
        int times=3;
        int i=0;
        int counter=0;
        int iteration=5;

        while (i<iteration)
        {

        times=delay1(times);
                counter+=4;
                i++;
        }
}
```

```
int delay1 (int i)
{
        int j = 2;
        int k = 7;
        int l = 1;
        if (i>j)
        {
                i--;
                i = delay1(i);
        }
        return k*l;
}
```

C Program flow: Passing arguments between functions

```
main()
{
        int times=3;
        int i=0;
        int counter=0;
        int iteration=5;

        while (i<iteration)
        {

        times=delay1(times);
                counter+=4;
                i++;
        }
}
```

i = 3

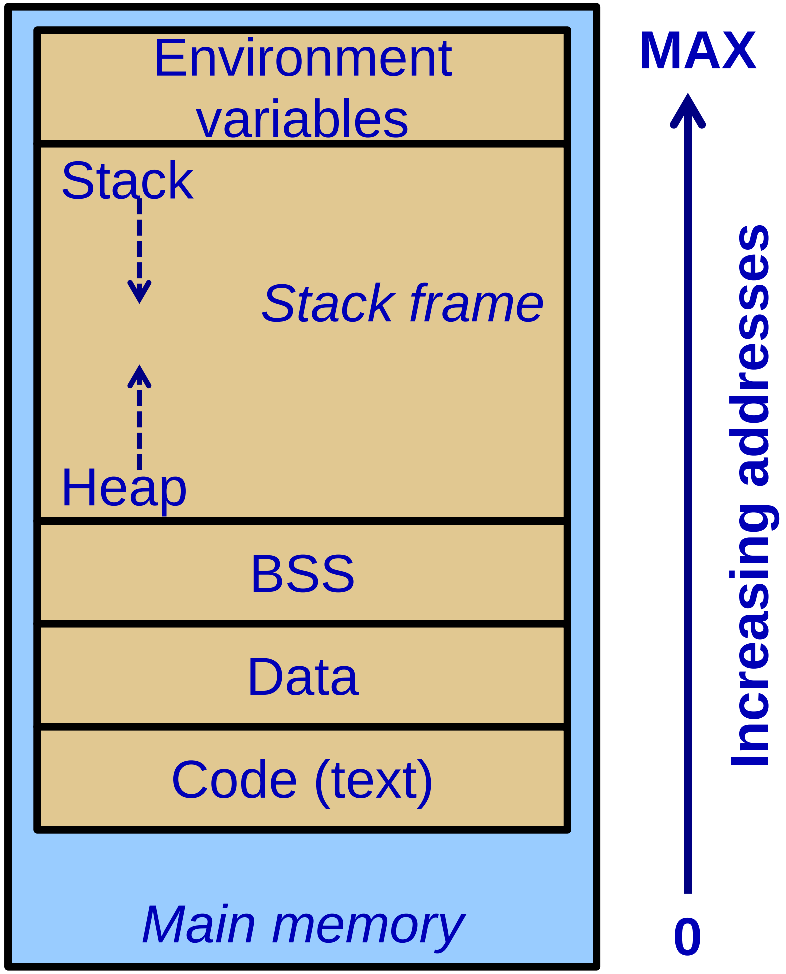
```
int delay1 (int i)
{
        int j = 2;
        int k = 7;
        int l = 1;
        if (i>j)
        {
                i--;
                i = de
        }
        return k*l;
}
```

i = 2

```
int delay1 (int i)
{
        int j = 2;
        int k = 7;
        int l = 1;
        if (i>j)
        {
           i--;
           i = delay1(i);
        }
        return k*l;
}
```

Memory regions in an ARM program

- A running program has 3 regions (both in C or in assembly)
  - Code
    - Code or instructions (text)
  - Data
    - Global data
    - BSS: data initialized to 0
  - Stack frame
    - Stack: automatic variables
    - Heap: explicitly managed dynamic variable



Memory regions in an ARM program: The memory stack

- Specific region of the memory managed as a stack of data
- Three main uses
  - **Placing local variables**
    - E.g., Variables *a*, *b*, *temp* and *i*
    - <u>*This will not be covered in the course*</u>
    - We have up to 8 general-purpose registers to store local variables

```c
int fibonacci(int n)
{
    int a = 1;
    int b = 1;
    int temp = 1, i;
        for(i = 0; i<n-2; i++)
        {
            temp = a + b;
            a = b;
            b = temp;
        }
        return temp;
}
```

  - **Passing arguments (Function call)**
    - Shared data between functions
  - **Store and retrieve register values**
    - Several general-purpose registers may be used during the execution of a function
    - The programmer must save the values of the registers before using them at the beginning of the function
    - The values must be restored at the end of the function

ARM function call procedure: Passing arguments

- In a function call, arguments are passed through registers
  - Up to 4 parameters: *r0-r3*
- The return value is stored in *r0* by the function

```c
int main(void)
{
    int c;
    c = fibonacci(5);

    while(1){
        swiWaitForVBlank();
    }
}
```

```
156              c = fibonacci(5);
02000440:    mov r0, #5        ; 0x5
02000444:    b  0x20003bc <fibonacci>
02000448:    mov r3, r0
```

The argument (#5) is store in *r0*

- Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …
  - $F_0 = 0$; $F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$

ARM function call procedure: Passing arguments

- In a function call, arguments are passed through registers
  - Up to 4 parameters: *r0-r3*
- The return value is stored in *r0* by the function.

```c
int main(void)
{
    int c;
    c = fibonacci(5);

    while(1){
        swiWaitForVBlank();
    }
}
```

```
156                c = fibonacci(5);
02000440:     mov r0, #5          ; 0x5
02000444:     bl 0x20003bc <fibonacci>
02000448:     mov r3, r0
```

Branch and link to the function.

It leaves the address `0x32000448` in *lr*

ARM function call procedure: Passing arguments

- In a function call, arguments are passed through registers
  - Up to 4 parameters: *r0-r3*
- The return value is stored in *r0* by the function.

```c
int main(void)
{
    int c;
    c = fibonacci(5);

    while(1){
        swiWaitForVBlank();
    }
}
```

```
156               c = fibonacci(5);
02000440:    mov r0, #5          ; 0x5
02000444:    bl 0x20003bc <fibonacci>
02000448:    mov r3, r0
```

The result (returned in *r0*) can be used.

ARM function call procedure: Passing arguments

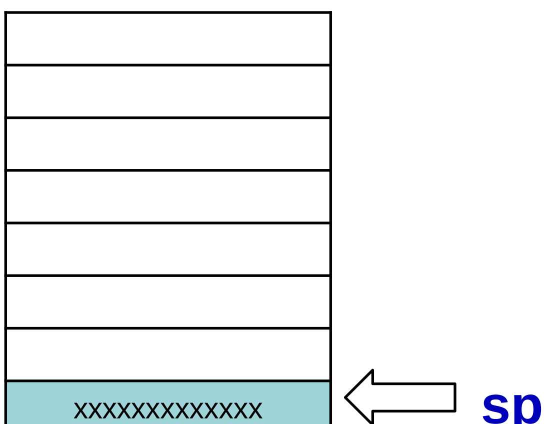
- If there are more than 4 arguments, the rest are passed through the stack

```c
int main(void)
{
    int c;
    c = megaFUnction(1,2,3,4,5,6);

    while(1){
        swiWaitForVBlank();
    }
}
```

```
02000494:   sub sp, sp, #20 ; 0x14
166           c = megaFUnction(1,2,3,4,5,6);
02000498    mov r3, #5      ; 0x5
0200049c    str r3, [sp]
020004a0    mov r3, #6      ; 0x6
020004a4    str r3, [sp, #4]
020004a8    mov r0, #1      ; 0x1
020004ac    mov r1, #2      ; 0x2
020004b0    mov r2, #3      ; 0x3
020004b4    mov r3, #4      ; 0x4
020004b8    bl 0x2000438 <megaFUnction>
020004bc    mov r3, r0
```

Some space is ‘reserved’ in the stack by moving the stack pointer



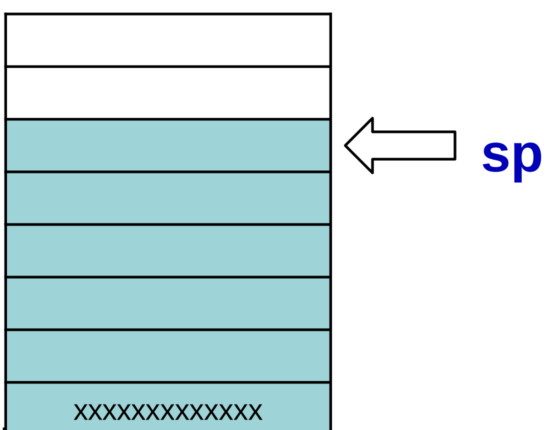
ARM function call procedure: Passing arguments

- If there are more than 4 arguments, the rest are passed through the stack

```c
int main(void)
{
    int c;
    c = megaFUnction(1,2,3,4,5,6);

    while(1){
        swiWaitForVBlank();
    }
}
```

```
02000494:    sub sp, sp, #20 ; 0x14
166            c = megaFUnction(1,2,3,4,5,6);
02000498     mov r3, #5      ; 0x5
0200049c     str r3, [sp]
020004a0     mov r3, #6      ; 0x6
020004a4     str r3, [sp, #4]
020004a8     mov r0, #1      ; 0x1
020004ac     mov r1, #2      ; 0x2
020004b0     mov r2, #3      ; 0x3
020004b4     mov r3, #4      ; 0x4
020004b8     bl 0x2000438 <megaFUnction>
020004bc     mov r3, r0
```



Some space is ‘reserved’ in the stack by moving the stack pointer

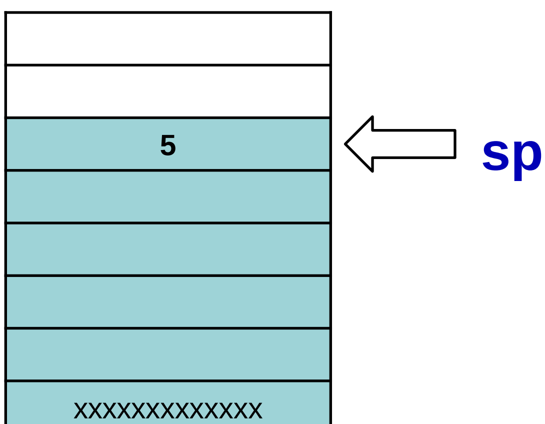
ARM function call procedure: Passing arguments

- If there are more than 4 arguments, the rest are passed through the stack

```c
int main(void)
{
    int c;
    c = megaFUnction(1,2,3,4,5,6);

    while(1){
        swiWaitForVBlank();
    }
}
```

```
02000494:   sub sp, sp, #20 ; 0x14
166           c = megaFUnction(1,2,3,4,5,6);
02000498:   mov r3, #5       ; 0x5
0200049c:   str r3, [sp]
020004a0:   mov r3, #6       ; 0x6
020004a4    str r3, [sp, #4]
020004a8    mov r0, #1       ; 0x1
020004ac    mov r1, #2       ; 0x2
020004b0    mov r2, #3       ; 0x3
020004b4    mov r3, #4       ; 0x4
020004b8    bl 0x2000438 <megaFUnction>
020004bc    mov r3, r0
```



The 5th argument is stored in the stack

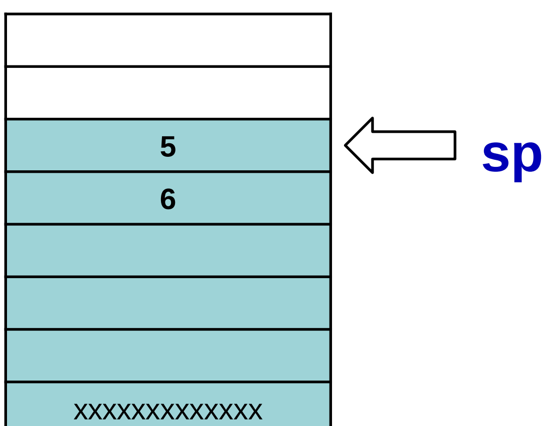
ARM function call procedure: Passing arguments

- If there are more than 4 arguments, the rest are passed through the stack

```c
int main(void)
{
    int c;
    c = megaFUnction(1,2,3,4,5,6);

    while(1){
        swiWaitForVBlank();
    }
}
```

```
02000494:    sub sp, sp, #20 ; 0x14
166            c = megaFUnction(1,2,3,4,5,6);
02000498:    mov r3, #5       ; 0x5
0200049c:    str r3, [sp]
020004a0:    mov r3, #6       ; 0x6
020004a4:    str r3, [sp, #4]
020004a8     mov r0, #1       ; 0x1
020004ac     mov r1, #2       ; 0x2
020004b0     mov r2, #3       ; 0x3
020004b4     mov r3, #4       ; 0x4
020004b8     bl 0x2000438 <megaFUnction>
020004bc     mov r3, r0
```



The 6th argument is stored in the stack

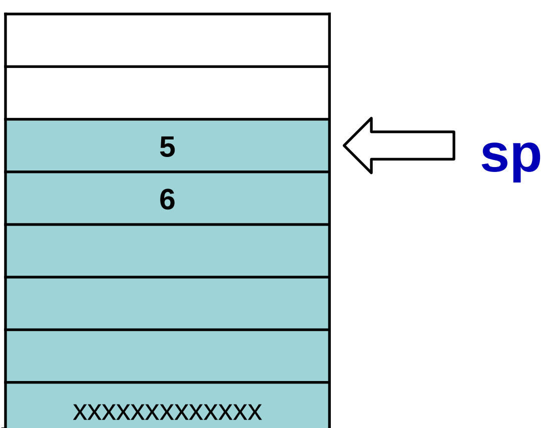
ARM function call procedure: Passing arguments

- If there are more than 4 arguments, the rest are passed through the stack

```c
int main(void)
{
    int c;
    c = megaFUnction(1,2,3,4,5,6);

    while(1){
        swiWaitForVBlank();
    }
}
```

```
02000494:    sub sp, sp, #20 ; 0x14
166            c = megaFUnction(1,2,3,4,5,6);
02000498:    mov r3, #5       ; 0x5
0200049c:    str r3, [sp]
020004a0:    mov r3, #6       ; 0x6
020004a4:    str r3, [sp, #4]
020004a8:    mov r0, #1       ; 0x1
020004ac:    mov r1, #2       ; 0x2
020004b0:    mov r2, #3       ; 0x3
020004b4:    mov r3, #4       ; 0x4
020004b8:    bl 0x2000438 <megaFUnction>
020004bc:    mov r3, r0
```



First 4 arguments are passed through registers r0, r1, r2 and r3

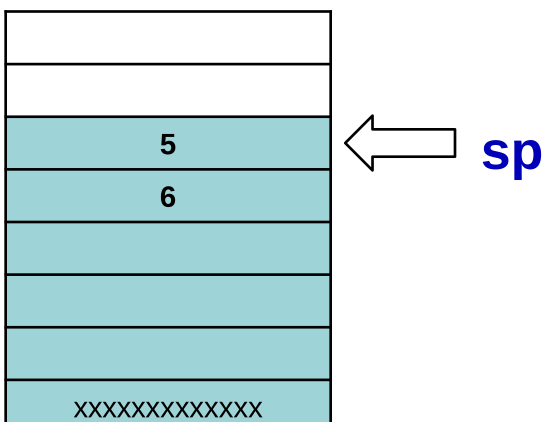
ARM function call procedure: Passing arguments

- If there are more than 4 arguments, the rest are passed through the stack

```c
int main(void)
{
    int c;
    c = megaFUnction(1,2,3,4,5,6);

    while(1){
        swiWaitForVBlank();
    }
}
```

```
02000494:   sub sp, sp, #20 ; 0x14
166           c = megaFUnction(1,2,3,4,5,6);
02000498:   mov r3, #5      ; 0x5
0200049c:   str r3, [sp]
020004a0:   mov r3, #6      ; 0x6
020004a4:   str r3, [sp, #4]
020004a8:   mov r0, #1      ; 0x1
020004ac:   mov r1, #2      ; 0x2
020004b0:   mov r2, #3      ; 0x3
020004b4:   mov r3, #4      ; 0x4
020004b8:   bl 0x2000438 <megaFUnction>
020004bc:   mov r3, r0
```



Branch and link to the function. It leaves the address **0x020004bc** in *lr*

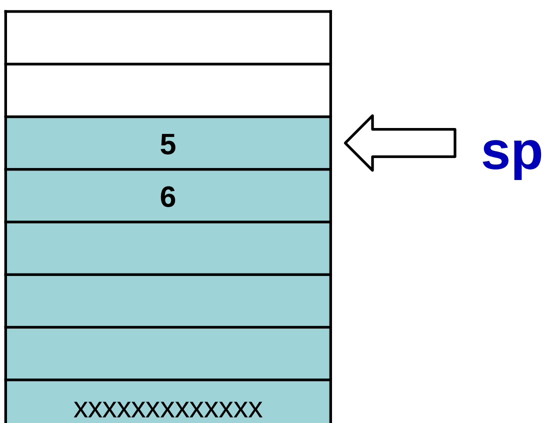
ARM function call procedure: Passing arguments

- If there are more than 4 arguments, the rest are passed through the stack

```c
int main(void)
{
    int c;
    c = megaFUnction(1,2,3,4,5,6);

    while(1){
        swiWaitForVBlank();
    }
}
```

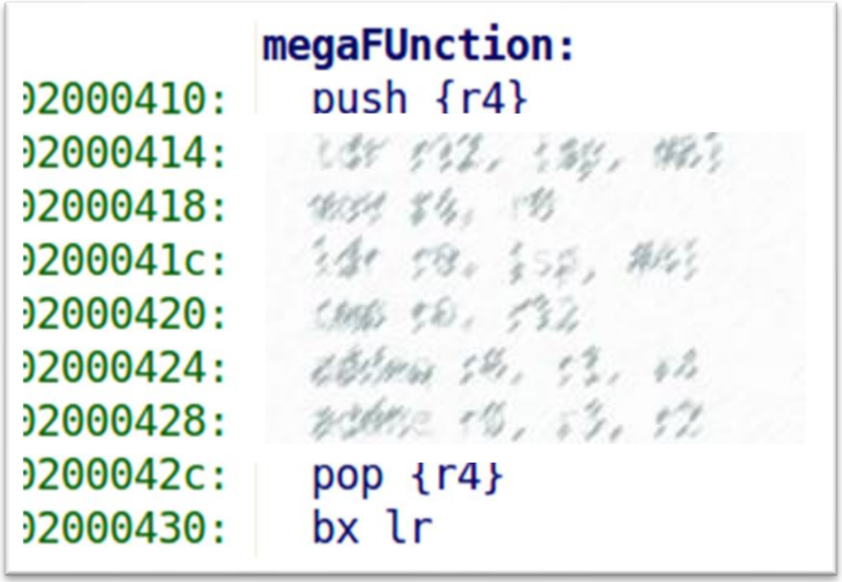
```
02000494:   sub sp, sp, #20 ; 0x14
166           c = megaFUnction(1,2,3,4,5,6);
02000498:   mov r3, #5      ; 0x5
0200049c:   str r3, [sp]
020004a0:   mov r3, #6      ; 0x6
020004a4:   str r3, [sp, #4]
020004a8:   mov r0, #1      ; 0x1
020004ac:   mov r1, #2      ; 0x2
020004b0:   mov r2, #3      ; 0x3
020004b4:   mov r3, #4      ; 0x4
020004b8:   bl 0x2000438 <megaFUnction>
020004bc:   mov r3, r0
```

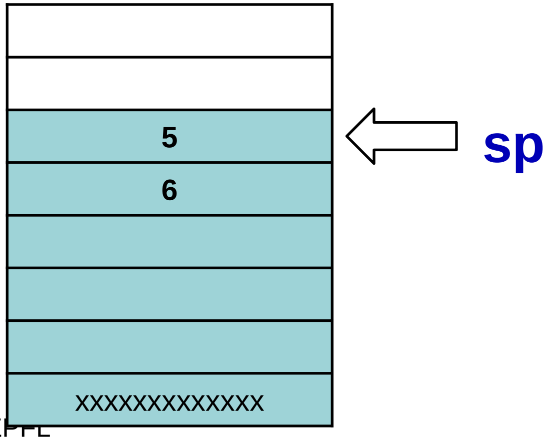


The result (returned in ***r0***) can be used

ARM function call procedure: Saving and retrieving registers

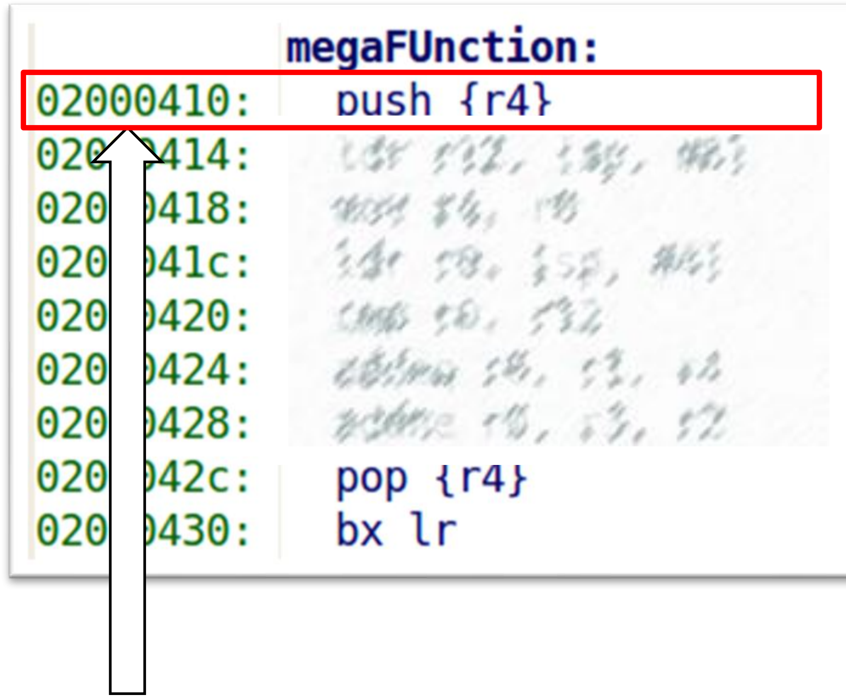
- The functionality of the function is unknown but register *r4* is used inside.
  - *r4* must be saved in the stack
  - The value is pushed
  - *sp* is updated.

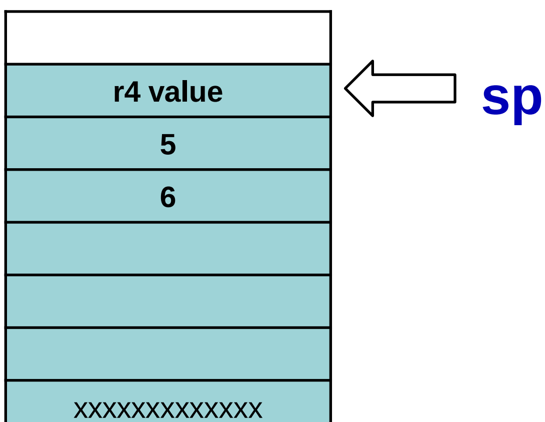




ARM function call procedure: Saving and retrieving registers

- The functionality of the function is unknown but register *r4* is used inside.
  - *r4* must be saved in the stack
  - The value is pushed
  - *sp* is updated.



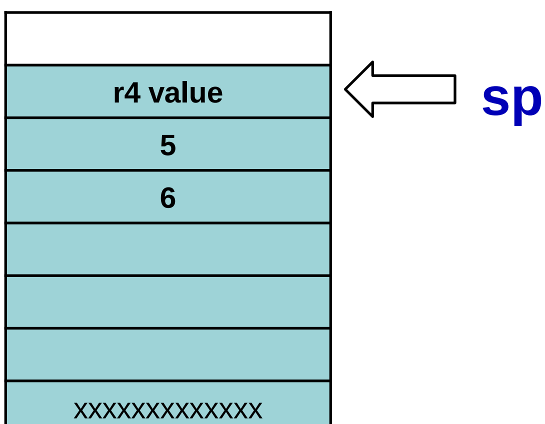


r4 is pushed and the *sp* updated

ARM function call procedure: Saving and retrieving registers

- The functionality of the function is unknown but register *r4* is used inside.
  - *r4* must be saved in the stack
  - The value is pushed
  - *sp* is updated.

```
          megaFUnction:
02000410:   push {r4}
02000414:
02000418:
0200041c:
02000420:
02000424:
02000428:
0200042c:   pop {r4}
02000430:   bx lr
```

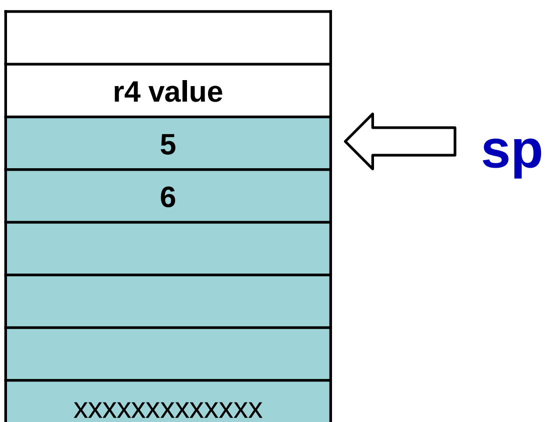


What happens if the previously stored values (arguments 5 and 6) need to be accessed?

ARM function call procedure: Saving and retrieving registers

- The functionality of the function is unknown but register *r4* is used inside.
  - *r4* must be saved in the stack
  - The value is pushed
  - *sp* is updated.

```
          megaFUnction:
02000410:   push {r4}
02000414:
02000418:
0200041c:
02000420:
02000424:
02000428:
0200042c:   pop {r4}
02000430:   bx lr
```



r4 is restored at the end of the execution and and the *sp* updated again

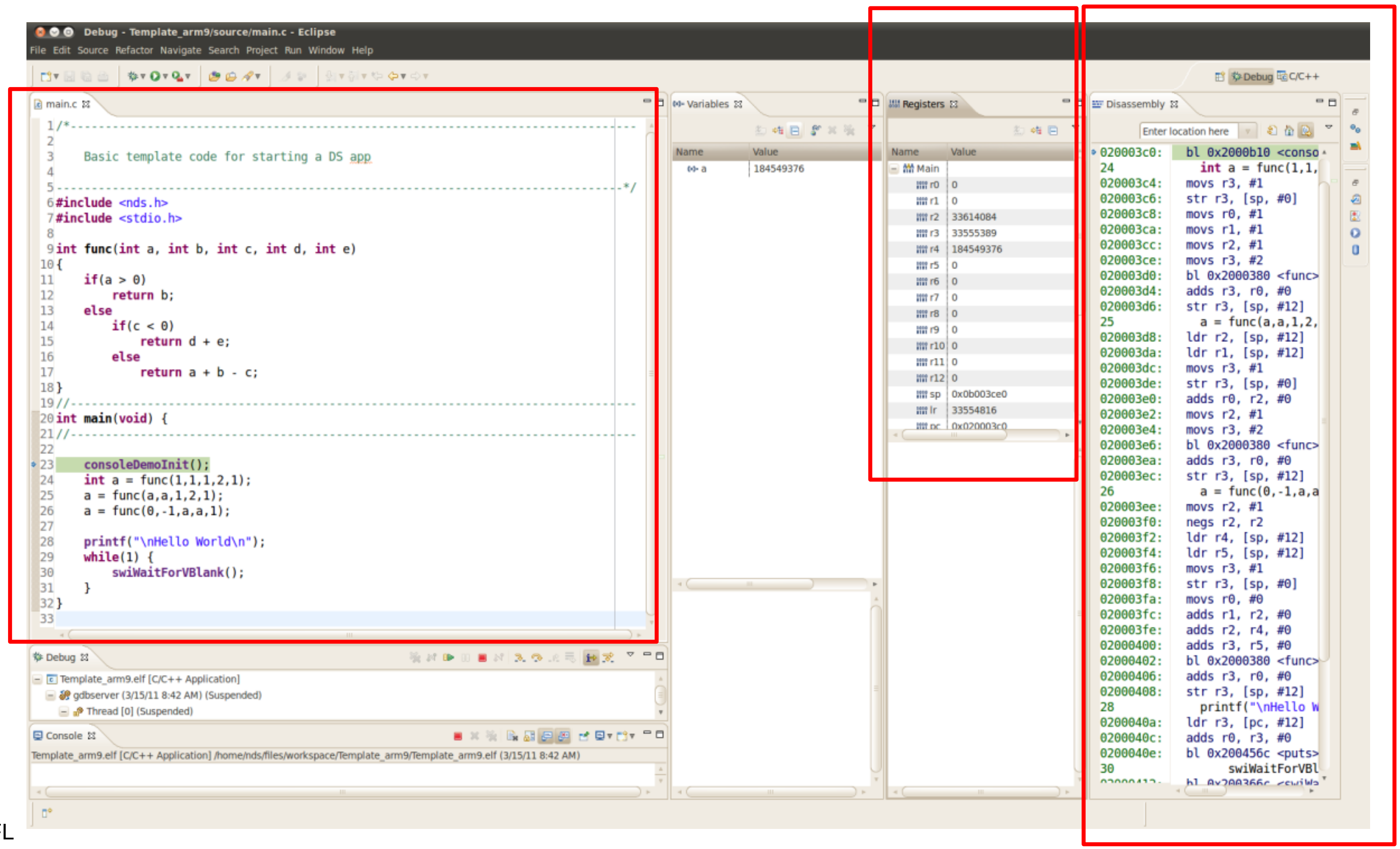
Practical Work 4: Introduction to ARM Assembly on the NDS

- Debugging assembly code
  - Multiple views needed, at least 3

Registers

Assembly

C



Practical Work 4: Introduction to ARM Assembly on the NDS

- Mixing C and ARM assembly

Inline Assembly

```
int a=6;
int b=4;
int c=0;
asm volatile("add %0, %1, %2" :
              "=r" (c) :
             "r" (a),"r" (b)
             );
```

To be implemented:

- Abs (a + b * c)

Stand alone .s file

```
.ARM
.ALIGN
.GLOBL  Sum
.TYPE   Sum,function
.text
Sum:

    stmdb sp!, {r4 - r11, lr}
    add r0, r1
    ldmia sp!, {r4 - r11, lr}
    bx lr
.end
```

To be implemented:

- Summation of array of elements
- Matrix multiplication

Practical Work 4: Introduction to ARM Assembly on the NDS

- Retrieving arguments from the stack
  - Implementing a function with 6 arguments
- C vs. Assembly code:
  - Performance comparison between manually written square root and a compiler generated one
- Improving performance using specific arithmetic instruction provided by ARMv5TE architecture
  - Optimizing a = a + b * c (multiply and accumulate or MAC instruct)

Practical Work 4: Introduction to ARM Assembly on the NDS

- Exercises (and homework)
  - Exercise 1 – Arithmetic operation with *inline* statement
  - Exercise 2 – Summation of arrays in a stand-alone .s file
  - Exercise 3 – Matrices multiplication in a stand-alone .s file
  - Exercise 4 – Retrieving arguments from the stack
  - Exercise 5 – Comparing performance
  - Exercise 6 – MAC application
  - *Exercise 7 – Matrix Multiplication Optimization
  - *Exercise 8 – Rounded Square Root

*Additional Exercises

Questions?

Let’s use ARM assembly and combine it with C in the NDS