Microcontroller Engineering TMIK13 Lecture 10

ASSEMBLER

ANDREAS AXELSSON (ANDREAS.AXELSSON@JU.SE)

Number Systems Recap

Base 10

```
uint16_t val = 46732;
```

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19

Base 2

```
uint16_t val = 0b1011011010001100;
```

0	1
10	11
100	101

Number Systems Recap

Decimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Base 2 – Binary Numbers

10110110_2

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	1	0	1	1	0
$1x2^7$	$0x2^6$	$1x2^5$	$1x2^4$	$0x2^3$	$1x2^2$	$1x2^1$	$0x2^0$

$128+0+32+16+0+4+2+0 = 182_{10}$

Base 16 – Hexadecimal Numbers

Base 16

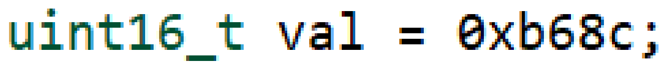
0 1 2 3 4 5 6 7 8 9 A B C D E F

Base 16 – Hexadecimal Numbers

Base 16

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

Example: 0xB68C



```
uint16_t val = 0xb68c;
```

16^3	16^2	16^1	16^0
B	6	8	C
$11x16^3$	$6x16^2$	$8x16^1$	$12x16^0$

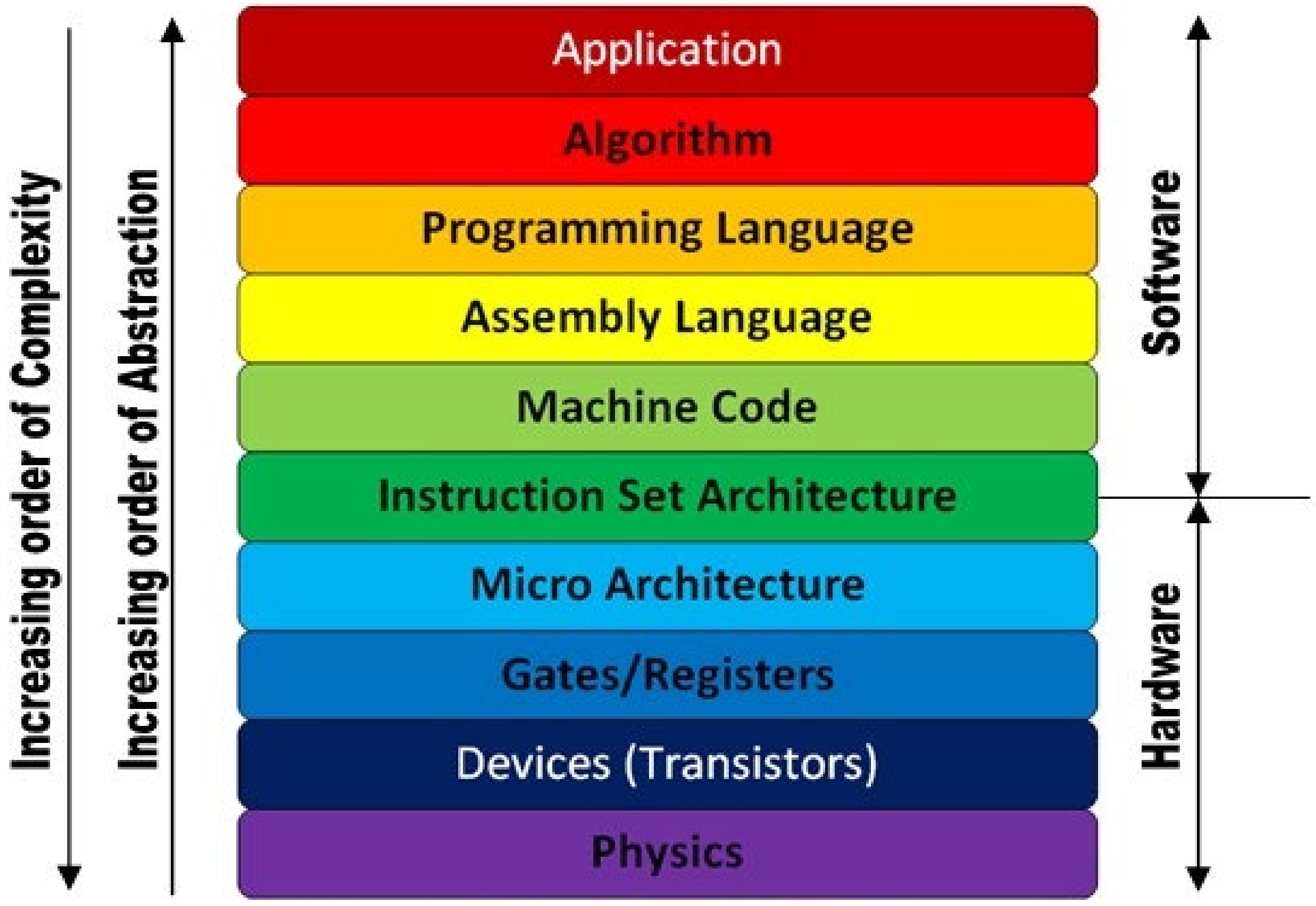
$$45056 + 1536 + 128 + 12 = 46732_{10}$$

Number Systems Recap

Decimal	Hexadecimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

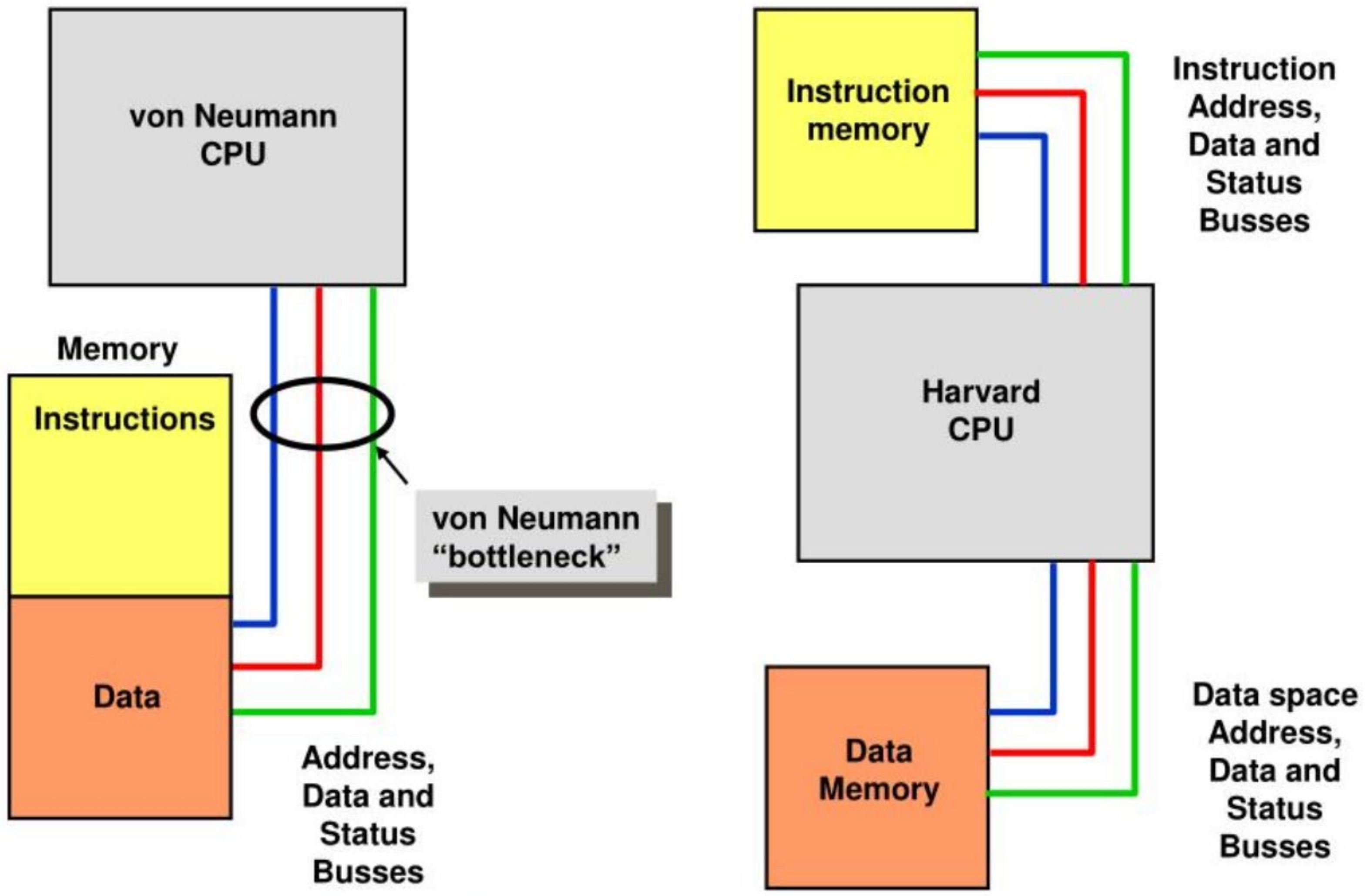
A group of four bits is called a ”Nibble”

Computer Architecture



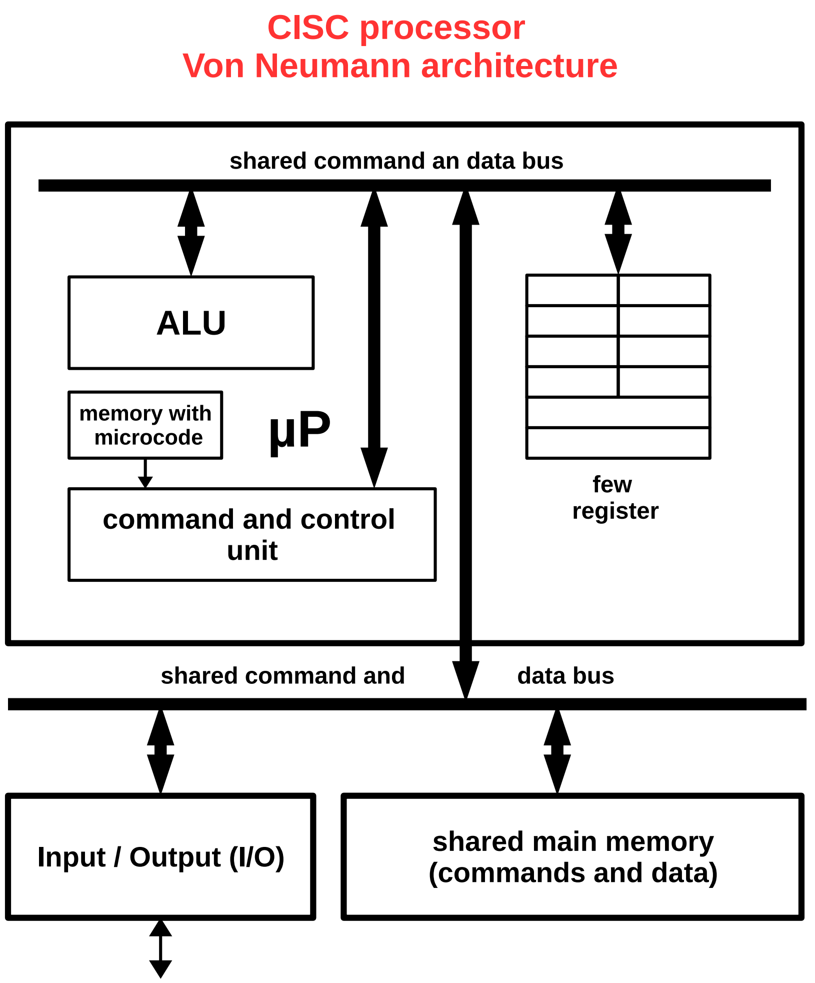
Layers of Abstraction

von Neumann vs Harvard CPU

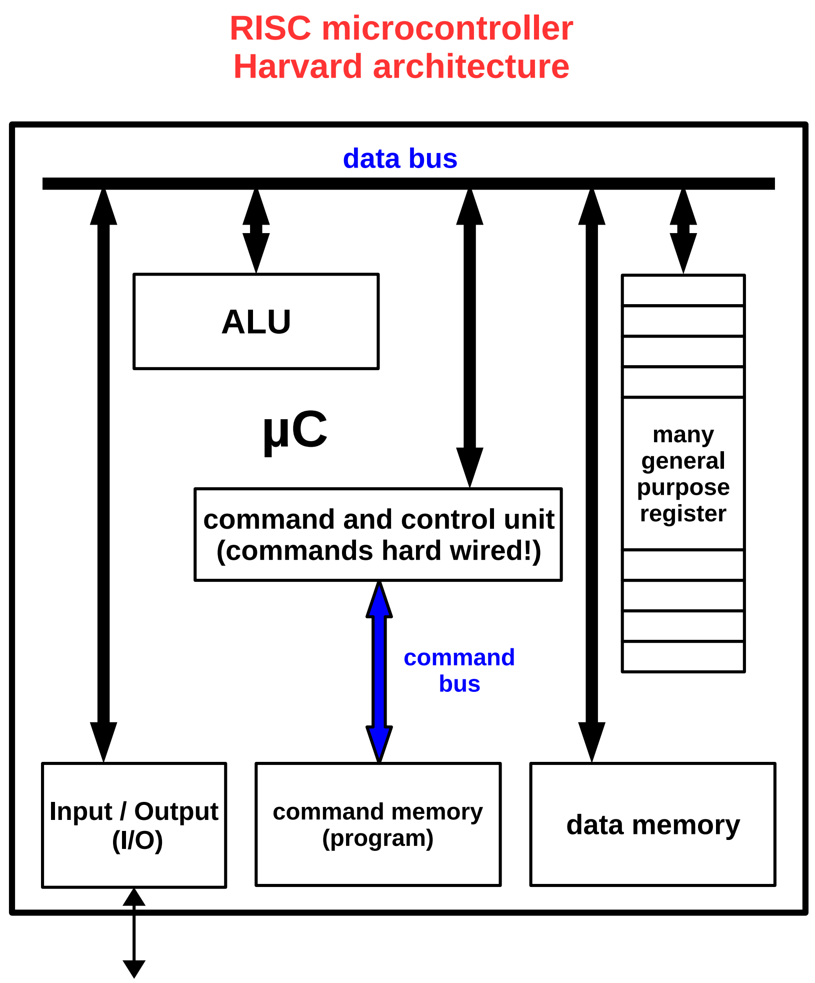

von Neumann CPU
Memory
Instructions
Data
von Neumann "bottleneck"
Address, Data and Status Busses
Instruction memory
Instruction Address, Data and Status Busses
Harvard CPU
Data Memory
Data space Address, Data and Status Busses

https://www.slideserve.com/sonia-alvarado/von-neumann-and-harvard-architectures

CISC vs RISC



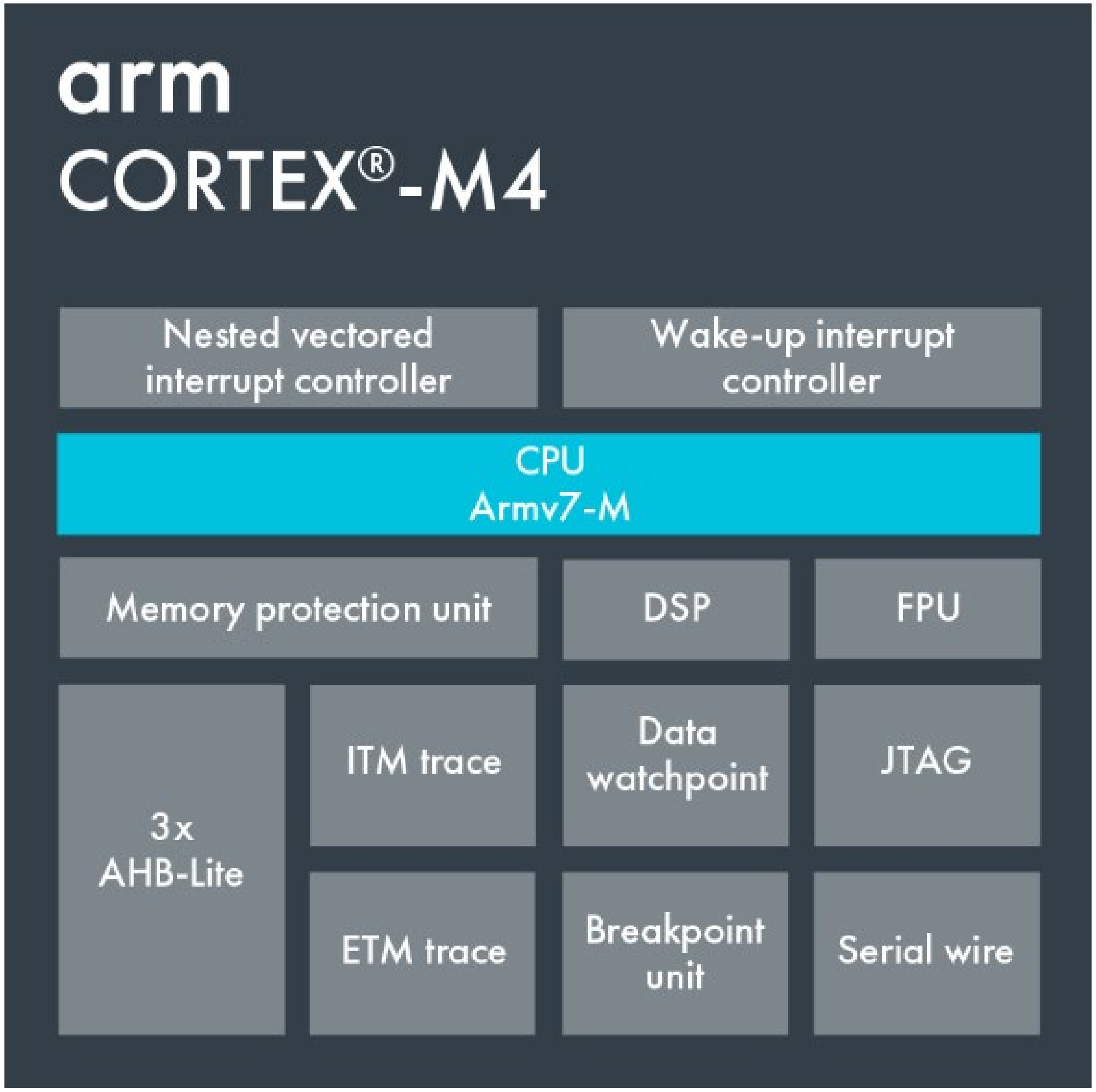
Complex Instruction Set Computer



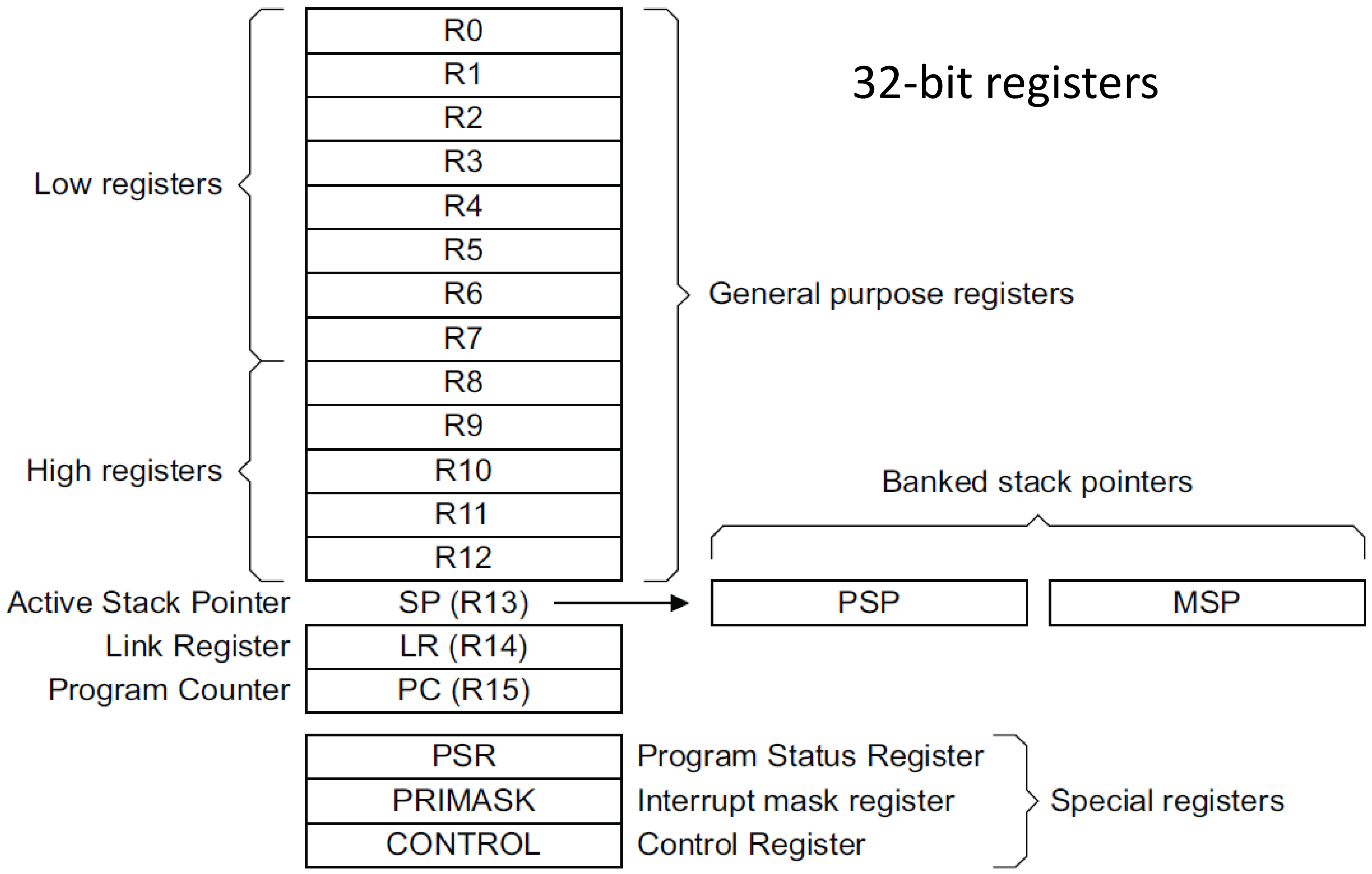
Reduced Instruction Set Computer

http://www.weigu.lu/tutorials/microcontroller/04_memory/index.html

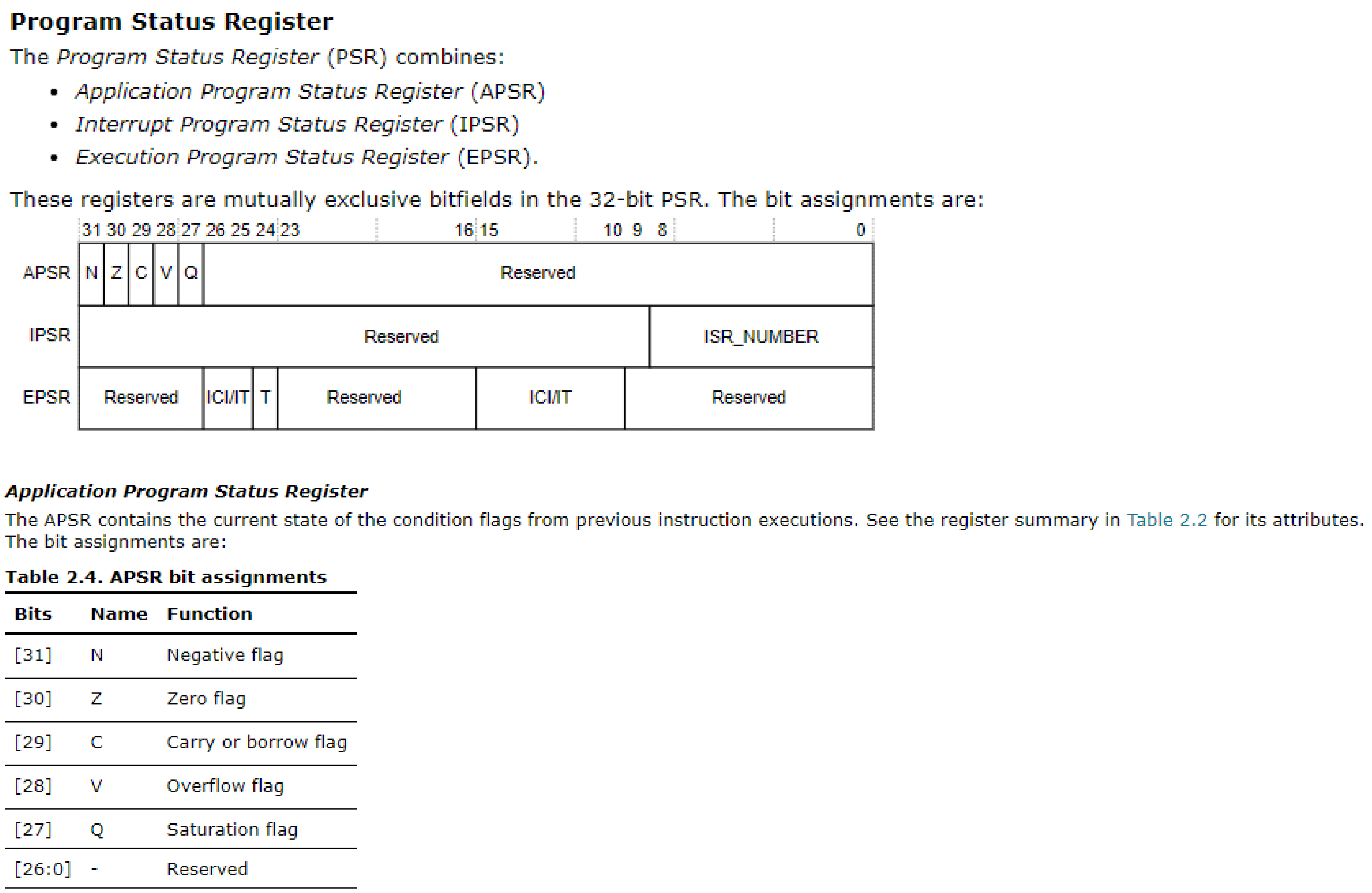
ARM Cortex-M4



ARM Processor Core Registers



ARM Processor Core Registers

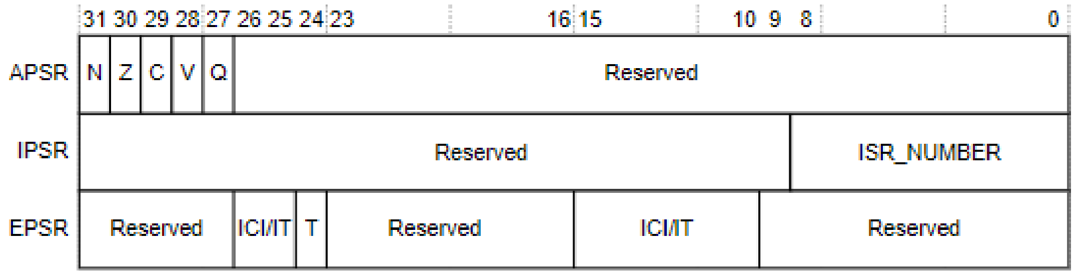


Program Status Register

The *Program Status Register* (PSR) combines:

- *Application Program Status Register* (APSR)
- *Interrupt Program Status Register* (IPSR)
- *Execution Program Status Register* (EPSR).

These registers are mutually exclusive bitfields in the 32-bit PSR. The bit assignments are:



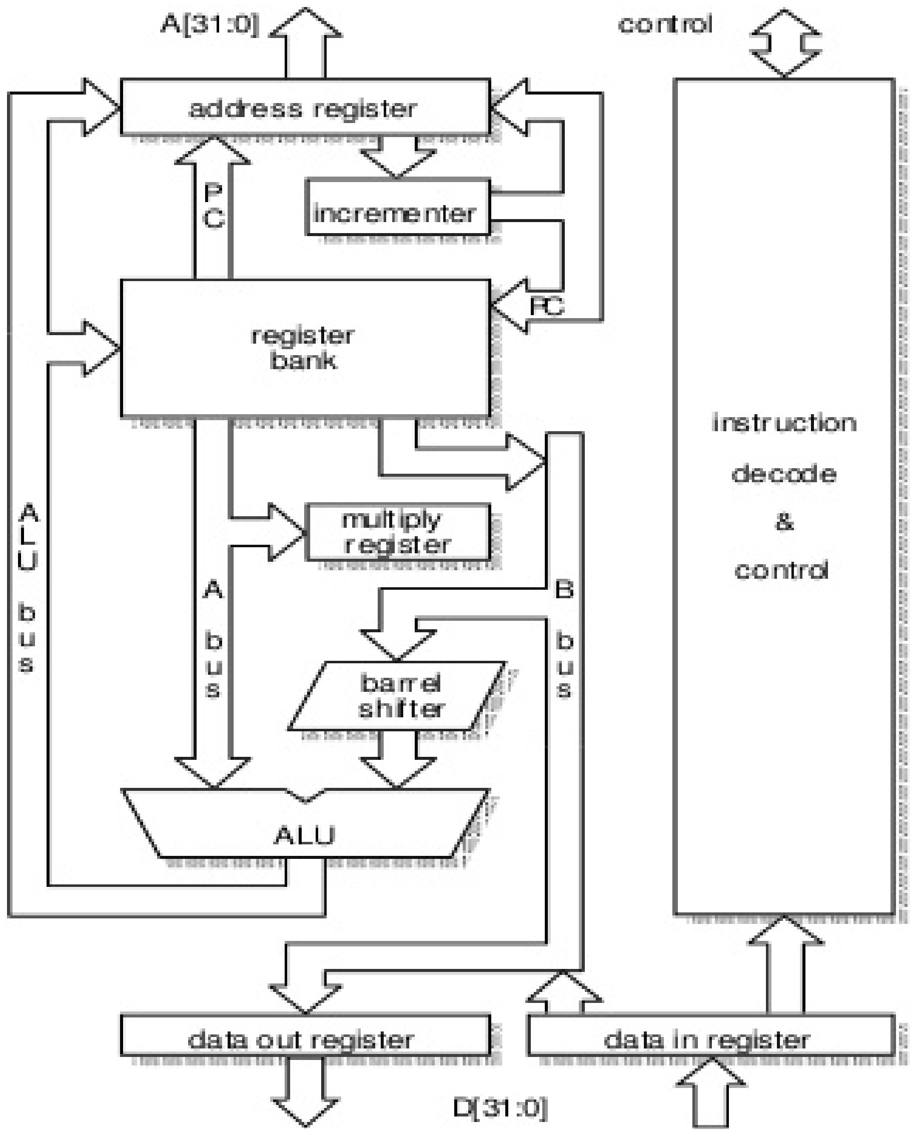
Application Program Status Register

The APSR contains the current state of the condition flags from previous instruction executions. See the register summary in Table 2.2 for its attributes. The bit assignments are:

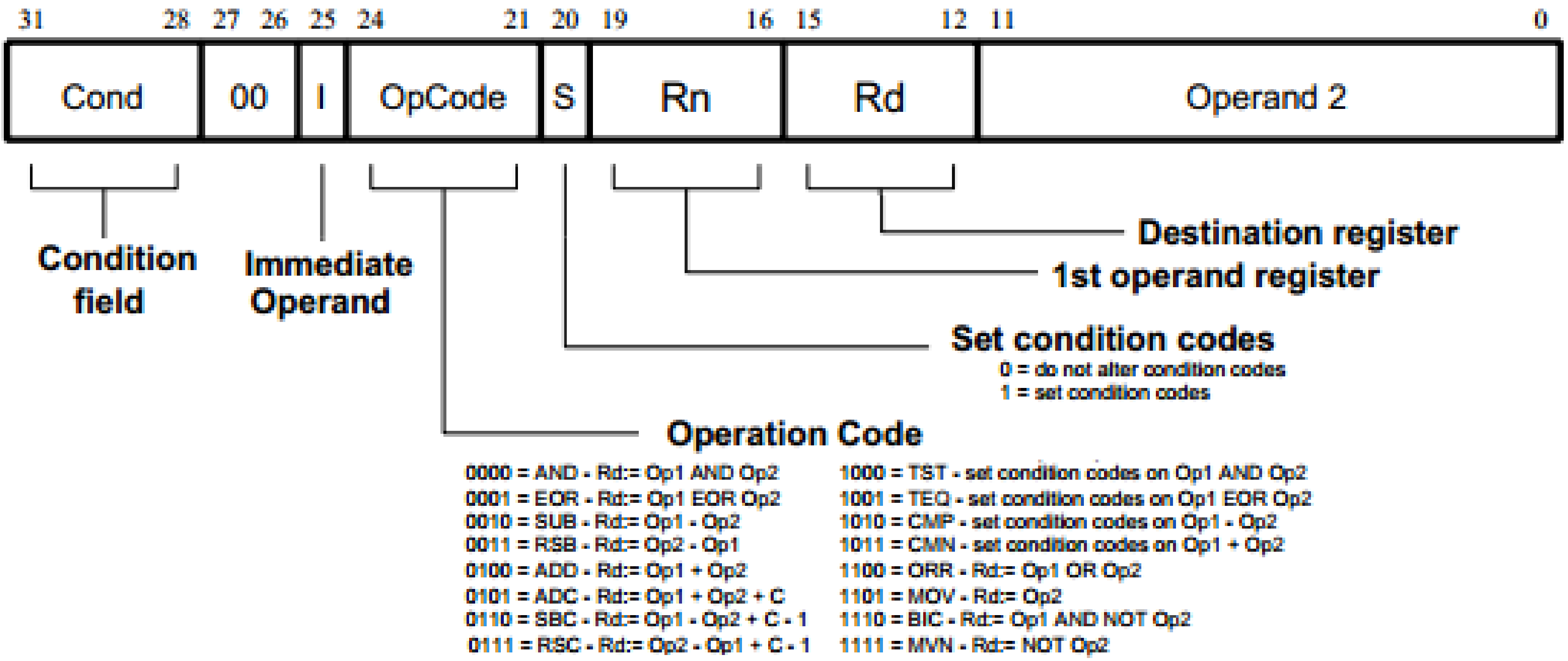
Table 2.4. APSR bit assignments

Bits	Name	Function
[31]	N	Negative flag
[30]	Z	Zero flag
[29]	C	Carry or borrow flag
[28]	V	Overflow flag
[27]	Q	Saturation flag
[26:0]	-	Reserved

ARM Architecture

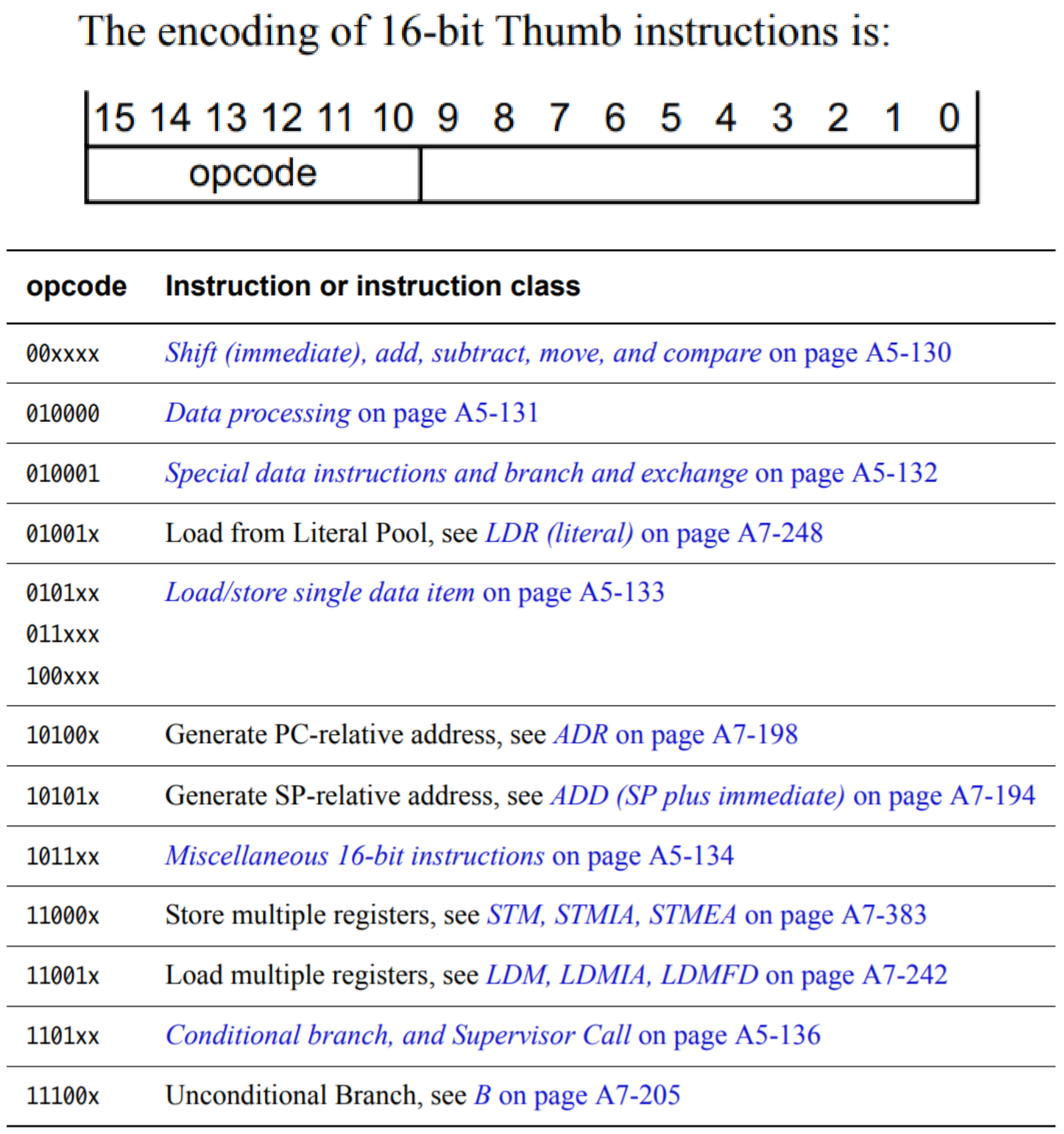


ARM Instruction Set Architecture



0000 = AND - Rd:= Op1 AND Op2	1000 = TST - set condition codes on Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2	1001 = TEQ - set condition codes on Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2	1010 = CMP - set condition codes on Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1	1011 = CMN - set condition codes on Op1 + Op2
0100 = ADD - Rd:= Op1 + Op2	1100 = ORR - Rd:= Op1 OR Op2
0101 = ADC - Rd:= Op1 + Op2 + C	1101 = MOV - Rd:= Op2
0110 = SBC - Rd:= Op1 - Op2 + C - 1	1110 = BIC - Rd:= Op1 AND NOT Op2
0111 = RSC - Rd:= Op2 - Op1 + C - 1	1111 = MVN - Rd:= NOT Op2

Thumb Instruction Set



The encoding of 16-bit Thumb instructions is:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode															

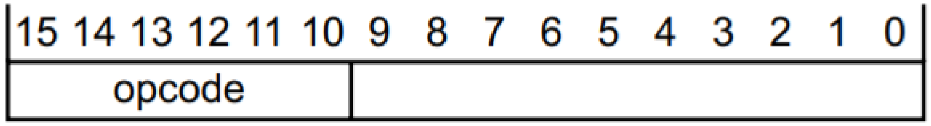
opcode	Instruction or instruction class
`00xxxx`	*Shift (immediate), add, subtract, move, and compare* on page A5-130
`010000`	*Data processing* on page A5-131
`010001`	*Special data instructions and branch and exchange* on page A5-132
`01001x`	Load from Literal Pool, see *LDR (literal)* on page A7-248

`0101xx`
`011xxx`
`100xxx`
	Load/store single data item on page A5-133

`10100x`	Generate PC-relative address, see *ADR* on page A7-198
`10101x`	Generate SP-relative address, see *ADD (SP plus immediate)* on page A7-194
`1011xx`	*Miscellaneous 16-bit instructions* on page A5-134
`11000x`	Store multiple registers, see *STM, STMIA, STMEA* on page A7-383
`11001x`	Load multiple registers, see *LDM, LDMIA, LDMFD* on page A7-242
`1101xx`	*Conditional branch, and Supervisor Call* on page A5-136
`11100x`	Unconditional Branch, see *B* on page A7-205

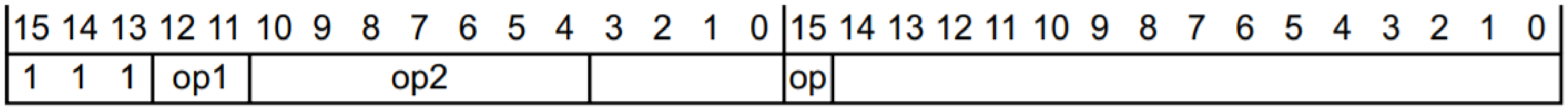
Thumb 2 Instruction Set

The encoding of 16-bit Thumb instructions is:



Special case if opcode is:
0b11101
0b11110
0b11111
Then instruction is a 32-bit Thumb instruction

The encoding of 32-bit Thumb instructions is:



Assembler Instruction Format

<operation> <operand1> <operand2> <operand3>

- There may be fewer operands
- First operand is typically destination (<Rd>)
- Other operands are sources (<Rn>, <Rm>)

Examples

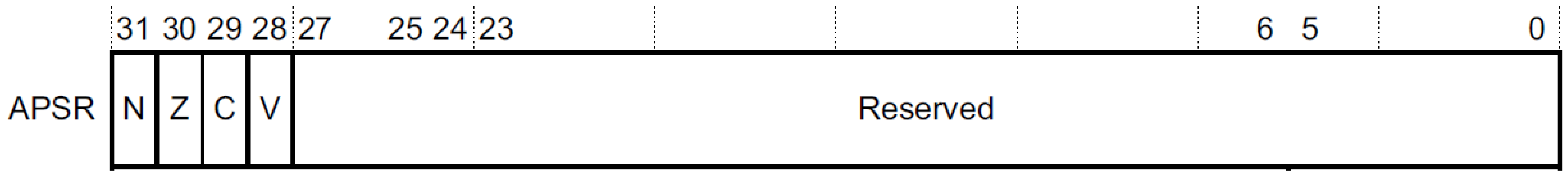
- ADDS <Rd>, <Rn>, <Rm>
  - Add registers: <Rd> = <Rn> + <Rm>
- AND <Rdn>, <Rm>
  - Bitwise and: <Rdn> = <Rdn> & <Rm>
- CMP <Rn>, <Rm>
  - Compare: Set condition flags based on result of computing <Rn> - <Rm>

Instruction Set Summary

Instruction Type	Instructions
Move	MOV
Load/Store	LDR, LDRB, LDRH, LDRSH, LDRSB, LDM, STR, STRB, STRH, STM
Add, Subtract, Multiply	ADD, ADDS, ADCS, ADR, SUB, SUBS, SBCS, RSBS, MULS
Compare	CMP, CMN
Logical	ANDS, EORS, ORRS, BICS, MVNS, TST
Shift and Rotate	LSLS, LSRS, ASRS, RORS
Stack	PUSH, POP
Conditional branch	IT, B, BL, B{cond}, BX, BLX
Extend	SXTH, SXTB, UXTH, UXTB
Reverse	REV, REV16, REVSH
Processor State	SVC, CPSID, CPSIE, SETEND, BKPT
No Operation	NOP
Hint	SEV, WFE, WFI, YIELD

”Mnemonics”

Update Condition Codes in APSR?



"S" suffix indicates the instruction updates APSR (program status register)

- ADD vs. ADDS
- ADC vs. ADCS
- SUB vs. SUBS
- MOV vs. MOVS

Load / Store Registers

ARM is a load/store architecture, so must process data in registers (not memory)

LDR: load register with word (32 bits) from memory

- LDR <Rt>, source address

STR: store register contents (32 bits) to memory

- STR <Rt>, destination address

Shift and Rotate

Common features

- All of these instructions update APSR condition flags
- Shift/rotate amount (in number of bits) specified by last operand

Logical shift left - shifts in zeroes on right

- LSLS <Rd>,<Rm>,#<imm5>
- LSLS <Rdn>,<Rm>

Logical shift right - shifts in zeroes on left

- LSRS <Rd>,<Rm>,#<imm5>
- LSRS <Rdn>,<Rm>

Arithmetic shift right - shifts in copies of sign bit on left (to maintain arithmetic sign)

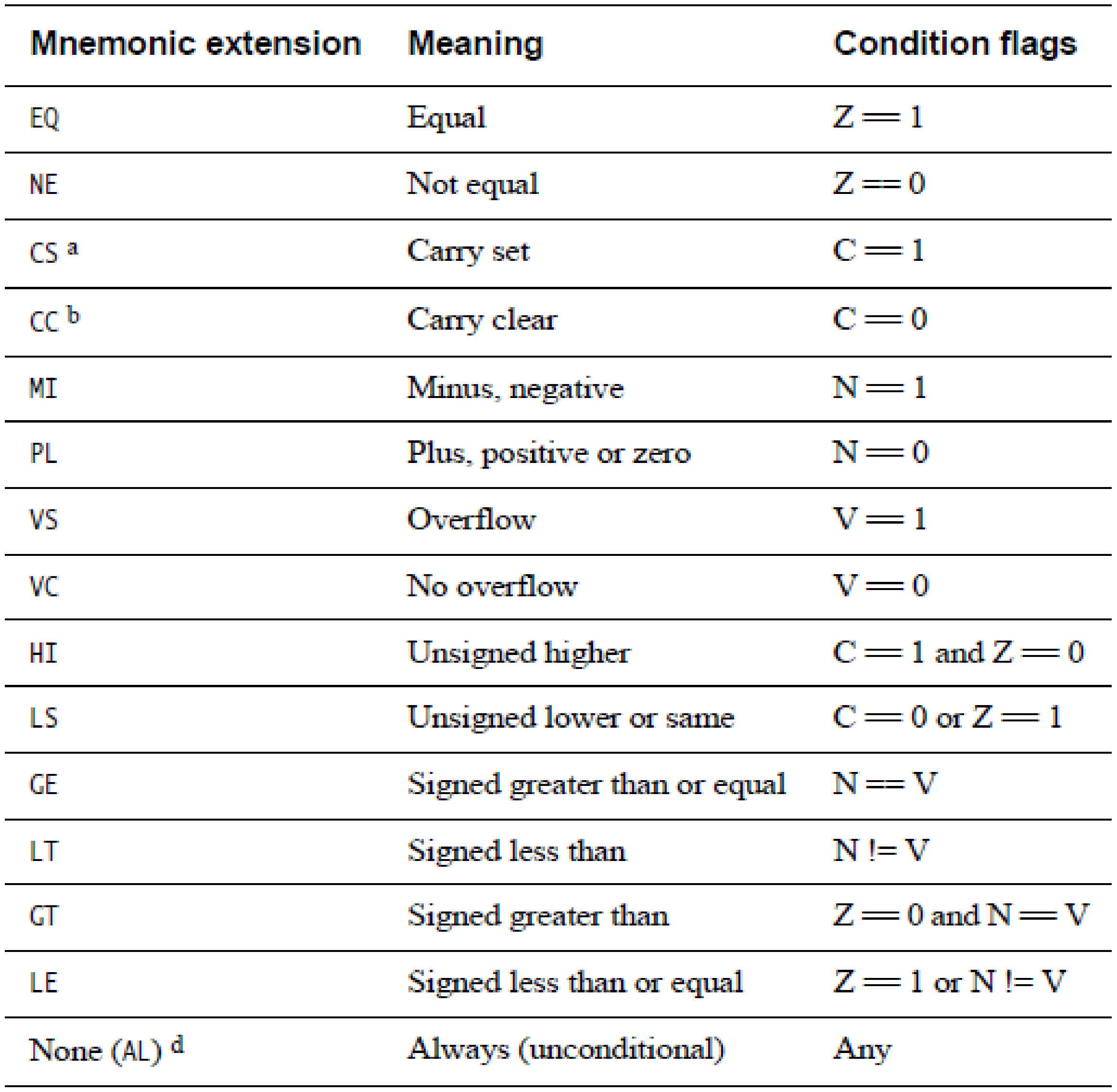
- ASRS <Rd>,<Rm>,#<imm5>

Rotate right

- RORS <Rdn>,<Rm>

Condition Codes

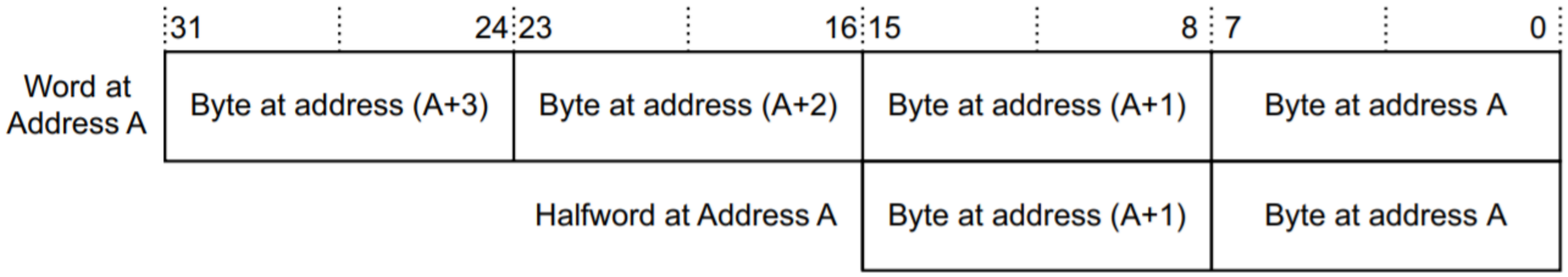
- Append to branch instruction (B) to make a conditional branch
- Full ARM instructions (not Thumb or Thumb-2) support conditional execution of arbitrary instructions
- Note: Carry bit = not-borrow for compares and subtractions



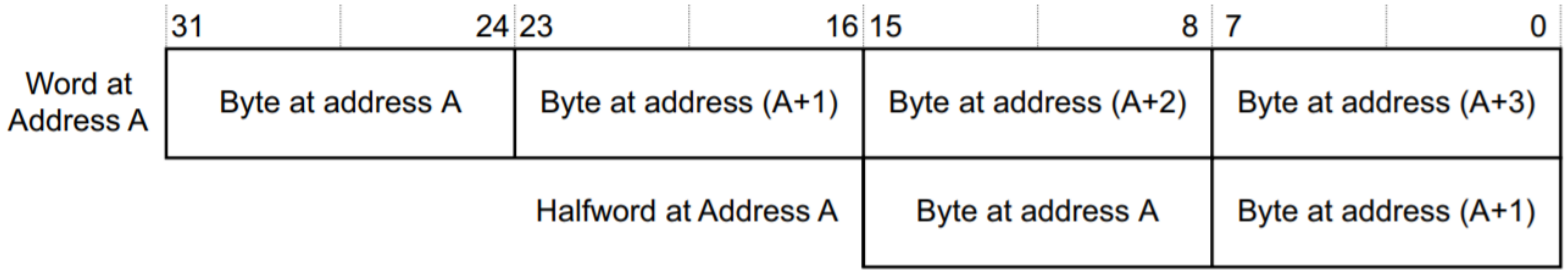
Mnemonic extension	Meaning	Condition flags
EQ	Equal	$Z == 1$
NE	Not equal	$Z == 0$
CS [a]	Carry set	$C == 1$
CC [b]	Carry clear	$C == 0$
MI	Minus, negative	$N == 1$
PL	Plus, positive or zero	$N == 0$
VS	Overflow	$V == 1$
VC	No overflow	$V == 0$
HI	Unsigned higher	$C == 1$ and $Z == 0$
LS	Unsigned lower or same	$C == 0$ or $Z == 1$
GE	Signed greater than or equal	$N == V$
LT	Signed less than	$N != V$
GT	Signed greater than	$Z == 0$ and $N == V$
LE	Signed less than or equal	$Z == 1$ or $N != V$
None (AL) [d]	Always (unconditional)	Any

Little Endian vs Big Endian

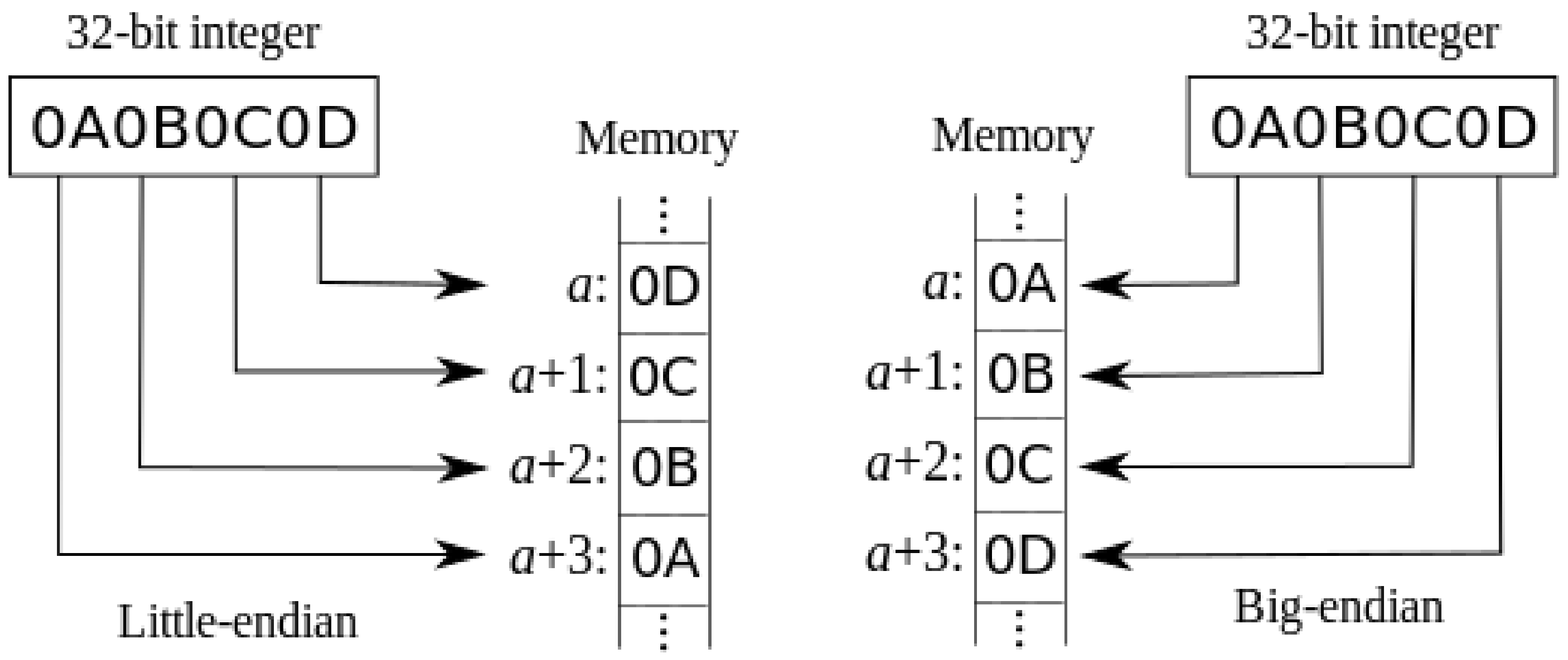
Little Endian



Big Endian



Little Endian vs Big Endian



https://en.wikipedia.org/wiki/Endianness

Simple program

```c
uint32_t mean;
uint32_t value_1;
uint32_t value_2;

uint32_t calc_mean(uint32_t v1, uint32_t v2)
{
    return (v1+v2)/2;
}
```

```
            calc_mean:
0x08000501:   push     {r7}
0x08000503:   sub      sp, #12
0x08000505:   add      r7, sp, #0
0x08000507:   str      r0, [r7, #4]
0x08000509:   str      r1, [r7, #0]
0x0800050b:   ldr      r2, [r7, #4]
0x0800050d:   ldr      r3, [r7, #0]
0x0800050f:   add      r3, r2
0x08000511:   lsrs     r3, r3, #1
0x08000513:   mov      r0, r3
0x08000515:   adds     r7, #12
0x08000517:   mov      sp, r7
0x08000519:   ldr.w    r7, [sp], #4
0x0800051d:   bx       lr
```

Main:

```
101              mean = calc_mean(value_1, value_2);
0x08000528:   ldr      r3, [pc, #680]  ; (0x80007d4 <main+692>)
0x0800052a:   ldr      r2, [r3, #0]
0x0800052c:   ldr      r3, [pc, #680]  ; (0x80007d8 <main+696>)
0x0800052e:   ldr      r3, [r3, #0]
0x08000530:   mov      r1, r3
0x08000532:   mov      r0, r2
0x08000534:   bl       0x8000500 <calc_mean>
0x08000538:   mov      r2, r0
0x0800053a:   ldr      r3, [pc, #672]  ; (0x80007dc <main+700>)
0x0800053c:   str      r2, [r3, #0]
```

Simple program

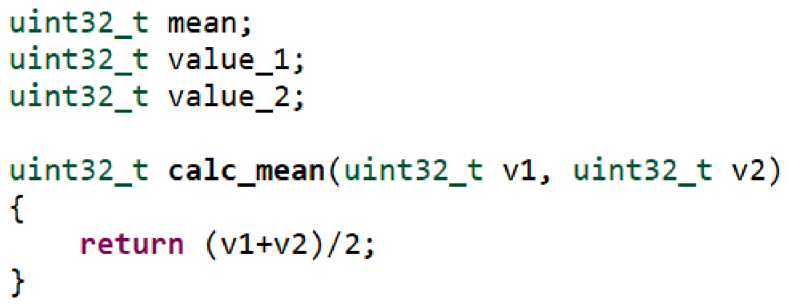
```c
uint32_t mean;
uint32_t value_1;
uint32_t value_2;

uint32_t calc_mean(uint32_t v1, uint32_t v2)
{
    return (v1+v2)/2;
}
```

```
                calc_mean:
0x08000501:     push    {r7}
0x08000503:     sub     sp, #12
0x08000505:     add     r7, sp, #0
0x08000507:     str     r0, [r7, #4]
0x08000509:     str     r1, [r7, #0]
0x0800050b:     ldr     r2, [r7, #4]
0x0800050d:     ldr     r3, [r7, #0]
0x0800050f:     add     r3, r2
0x08000511:     lsrs    r3, r3, #1
0x08000513:     mov     r0, r3
0x08000515:     adds    r7, #12
0x08000517:     mov     sp, r7
0x08000519:     ldr.w   r7, [sp], #4
0x0800051d:     bx      lr
```

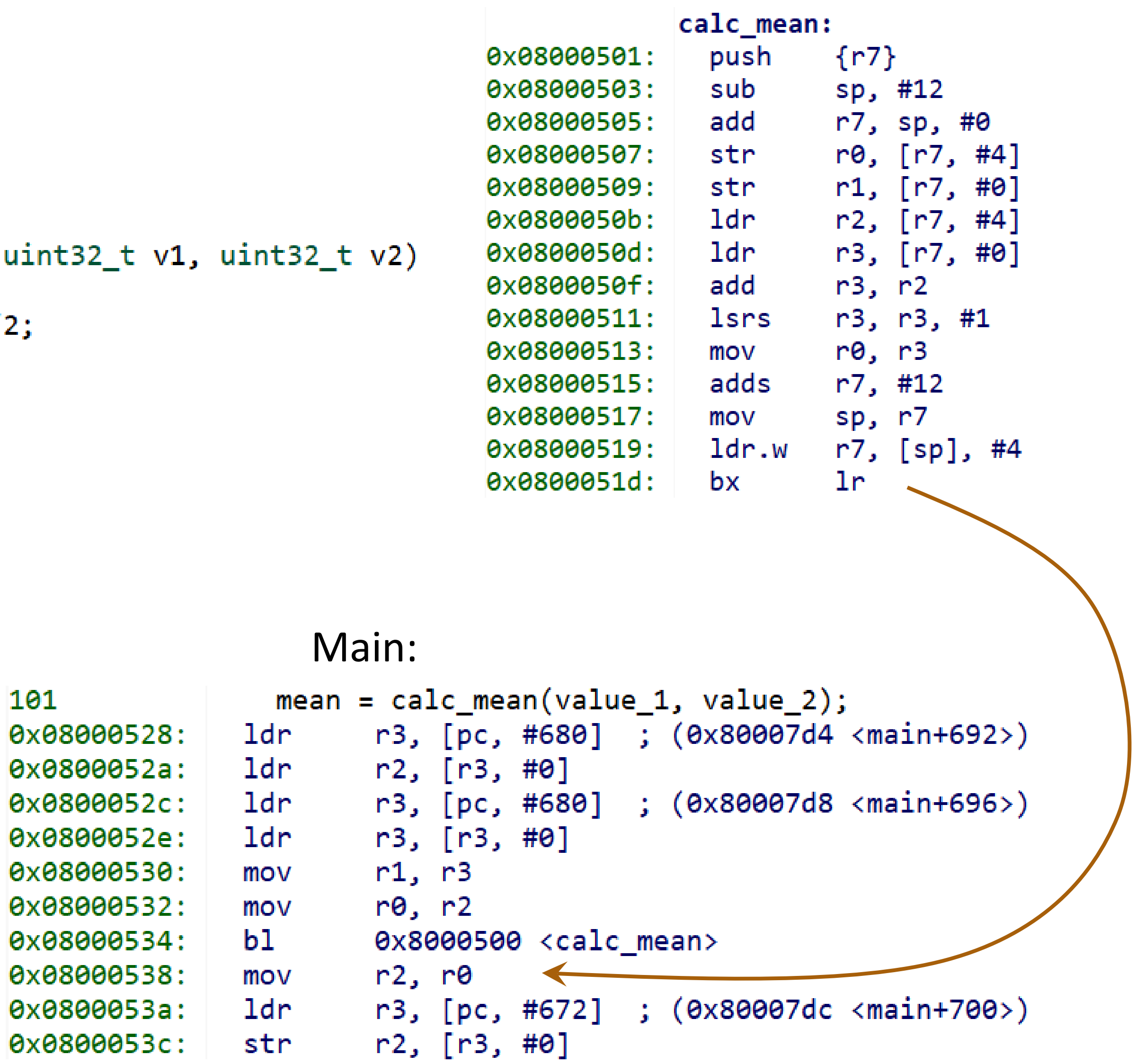
Main:

```
                  mean = calc_mean(value_1, value_2);
0x08000528:     ldr     r3, [pc, #680]  ; (0x80007d4 <main+692>)
0x0800052a:     ldr     r2, [r3, #0]
0x0800052c:     ldr     r3, [pc, #680]  ; (0x80007d8 <main+696>)
0x0800052e:     ldr     r3, [r3, #0]
0x08000530:     mov     r1, r3
0x08000532:     mov     r0, r2
0x08000534:     bl      0x8000500 <calc_mean>
0x08000538:     mov     r2, r0
0x0800053a:     ldr     r3, [pc, #672]  ; (0x80007dc <main+700>)
0x0800053c:     str     r2, [r3, #0]
```

Simple program



```c
uint32_t mean;
uint32_t value_1;
uint32_t value_2;

uint32_t calc_mean(uint32_t v1, uint32_t v2)
{
    return (v1+v2)/2;
}
```



```
              calc_mean:
0x08000501:     push    {r7}
0x08000503:     sub     sp, #12
0x08000505:     add     r7, sp, #0
0x08000507:     str     r0, [r7, #4]
0x08000509:     str     r1, [r7, #0]
0x0800050b:     ldr     r2, [r7, #4]
0x0800050d:     ldr     r3, [r7, #0]
0x0800050f:     add     r3, r2
0x08000511:     lsrs    r3, r3, #1
0x08000513:     mov     r0, r3
0x08000515:     adds    r7, #12
0x08000517:     mov     sp, r7
0x08000519:     ldr.w   r7, [sp], #4
0x0800051d:     bx      lr
```

Main:

```
                 mean = calc_mean(value_1, value_2);
0x08000528:   ldr     r3, [pc, #680]  ; (0x80007d4 <main+692>)
0x0800052a:   ldr     r2, [r3, #0]
0x0800052c:   ldr     r3, [pc, #680]  ; (0x80007d8 <main+696>)
0x0800052e:   ldr     r3, [r3, #0]
0x08000530:   mov     r1, r3
0x08000532:   mov     r0, r2
0x08000534:   bl      0x8000500 <calc_mean>
0x08000538:   mov     r2, r0
0x0800053a:   ldr     r3, [pc, #672]  ; (0x80007dc <main+700>)
0x0800053c:   str     r2, [r3, #0]
```

STM32CubeIDE Demo

Lets go!!!

Microcontroller Engineering

Questions?

Contact information

Andreas Axelsson

Email: andreas.axelsson@ju.se

Mobile: 0709-467760