Assembly

ARMv6-M / CORTEX-M0 / THUMB-1(LR LSB=1 always)

Assembler converts each Assembly instruction to 16-bit opcode

Instruction	Operands	Flags	Result	Bits
Extend				
SXTH	Rd, Rm		Rd = SignExt(Rm[7:0])	1011 0010 00xx xxx
SXTB	Rd, Rm		Rd = SignExt(Rm[15:0])	1011 0010 01xx xxx
UXTH	Rd, Rm		Rd = ZeroExt(Rm[7:0])	1011 0010 10xx xxx
UXTB	Rd, Rm		Rd = ZeroExt(Rm[15:0])	1011 0010 11xx xxx
Reverse				
REV,				
REV16,				
REVSH				
Branch (Store current PC in LR=> Return BX LR)				
B(C)(L)label	1110 0xxx xxxx xxxx		1101 xxxx xxxx xxxx	BL is 32 bits!! See later
B(L)X Rm	0100 0111 0xxx x000		<Without L, With>	0100 0111 1xxx x000
PROC/ENDP	FUNCTION / ENDFUNC, for debugger only, mark start&end			
Stack (Can handle LR&PC) LIFO Lowest register stored first (lowest address)				
PUSH	{registers}			1011 010LR xxxx xxxx
POP	{registers}			1011 110PC xxxx xxxx
Processor State				
BKPT,				
CPS,				
MRS / MSR	Rd, ASPR / ASPR, Rn		Rd = APSR / APSR = Rn, 32bit opcode	
SVC				
Hint / Synchronization				
SEV,				
DMB,				xxxx xxxx
DSB,				
ISB				
WFE,				
WFI,				
YIELD				
No Operation				
NOP				
Compare, only flags affected!				
CMP	Rn, Rm/imm8	N, Z, C, V	SUBS	0100 0010 10xx xxx 0100 0101 xxxx xxx 0010 1xxx xxxx xxx
CMN	Rn, Rm	N, Z, C, V	ADDS	0100 0010 11xx xxx
TST	Rn, Rm	N, Z	Logical AND	0100 0010 00xx xxx

ARMv6-M / CORTEX-M0 => Each Instruction 16-bit

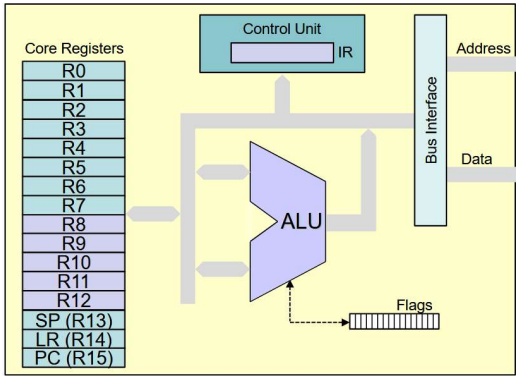
Written black Flags are **SET**, reds are **CLEARED** Only low registers(0-7)! OPCODE, EXT

Instruction	Operands	Flags	Result	Bits
MOVE				
MOV	Rd, Rm		Rd = Rm	0100 0110 xxxx xxx
MOVS	Rd, Rm/#imm8	N,Z; V	Rd = Rm	0000 0000 00xx xxx
EQU	MY_CONST8 EQU 0x12 => not stored in memory;; ^imm8			0010 0xxx xxxx xxx
Load/Store (Can handle SP, PC, LR) (offset /4 = imm8)				
LDR literal	Rt, [#imm8] ; = for bigger val		Rt = @(PC+imm)	0100 1xxx xxxx xxx
imm = imm8*4, offset always in words, align PC to next word **PSEUDO POSSIBLE**				
LDR imm5	Rt, [Rn, #imm5]		Rt=@(Rn+imm)	0110 1xxx xxxx xxx
LDR register	Rt, [Rn, Rm]		Rt=@(Rn+Rm)	0101 100x xxxx xxx
LDR(B110/H101) Same as LDR but only loads first byte/h.word, rest is filled with 0, **no literal**				
LDR(SB011/SH111) Instead of 0 extend, it extends signed, meaning it extends with MSB				
LDM	Rn{!}, {Rlist}	Starting address in Rn, !=auto incr.		1100 1xxx xxxx xxx
Loads 32-bit words from Rn into all entries in Rlist in order, ! is required if Rn in Rlist				
STR offs.	Rt, [Rn, #imm5]		@(Rn+imm) = Rt	0110 0xxx xxxx xxx
STR register	Rt, [Rn, Rm]		@(Rn+Rm) = Rt	0101 000x xxxx xxx
STRB(010)/STRH(001) Load only first Byte/Half word. NO zero extend				offsB(011|1|0)H(1000|0)
STM	Rn{!}, {Rlist}	Stores Rlist into Rn, "Reverse" LDM		1100 0xxx xxxx xxx
DC(B/W/D)	Store Byte,Half-word or Word in Memory => myLit DCD 0xFFEEDDCC			
Add, Subtract, Multiply				
ADD	Rdn, Rm		Rdn = Rdn + Rm	0100 0100 xxxx xxx
ADDS	Rd, Rn, Rm/#i3	N, Z, C, V	Rd = Rn + Rm/imm3	0001 100x xxxx xxx
ADDS imm8	Rd, #imm8	N, Z, C, V	Rd = Rd + imm8	0011 0xxx xxxx xxx
ADCS	Rdn, Rm	N, Z, C, V	Rdn = Rdn + Rm + C	0100 0001 01xx xxx
SUB	SP, SP, #imm7*4		SP = SP – imm7	1011 0000 1xxx xxx
SUBS	Rd, Rn, Rm/#i3	N, Z, C, V	Rd = Rn – Rm/imm3	0001 101x xxxx xxx
SUBS imm8	Rdn, #imm8	N, Z, C, V	Rdn = Rdn – imm8	0011 1xxx xxxx xxx
SBCS	Rdn, Rm	N, Z, C, V	Rdn = Rdn – Rm – NOT(C)	0100 0001 10xx xxx
RSBS	Rd, (Rn), #0	N, Z, C, V	Rd = #0 - Rn	0100 0010 01xx xxx
MULS	Rdm, Rn, Rdm	N, Z, C, V	Rdm = Rn * Rdm	0100 0011 01xx xxx
Logical				
ANDS	Rdn, (Rdn), Rm	N, Z	Rdn = Rdn & Rm => AND	0100 0000 00xx xxx
EORS	Rdn, (Rdn), Rm	N, Z	Rdn = Rdn $ Rm => XOR	0100 0000 01xx xxx
ORRS	Rdn, Rdn, Rm	N, Z	Rdn = Rdn # Rm => OR	0100 0011 00xx xxx
BICS	Rdn, Rdn, Rm	N, Z	Rdn = Rdn & !Rm => Clear	0100 0011 10xx xxx
MVNS	Rd, Rm	N, Z	Rd = !Rm => Bitwise NOT	0100 0011 11xx xxx
Shift and Rotate (LSLS C is unaffected if imm5 = 0, LSRS&ASRS not allowed) (for Rm bits)				
LSLS (shift)	Rdn, Rdn, Rm	N, Z, C	Rdn left, fill with 0	0100 0000 10xx xxx
LSRS (shift)	Rdn, Rdn, Rm	N, Z, C	Rdn right, fill with 0	0100 0000 11xx xxx
ASRS (shift)	Rdn, Rdn, Rm	N, Z, C	Rdn right, fill with MSB	0100 0001 00xx xxx
ALL ABOVE^	Rd, Rm, #imm5	N, Z, C	Do operation by imm5 bits	0000 0xxx xxxx xxx
RORS	Rdn, Rdn, Rm	N, Z, C	cyclic rotate right	0100 0001 11xx xxx

Cortex

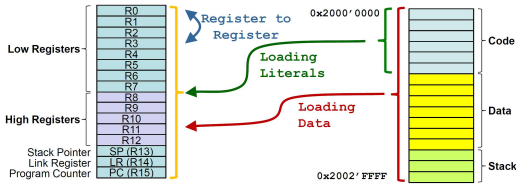
CPU

Core Registers: 16 Core Registers each 32-bit wide. Low R0-R7, high R8-R12, **Stack pointer** R13(LIFO for temporary data storage), **Link Register** R14(return address during function call), **Program Counter** R15(Address of next instruction)
Arithmetic Logic Unit: 32-bit wide does: Integer arithmetic, logic operations and shift/rotate
Flag-Register APSR: Bits based on results in ALU
Control Unit: Directs the flow of data between registers, ALU, memory, and I/O. Instruction Register (IR) = current opcode. Generates control signal
Bus Interface: Interface between int and ext. bus



Memory Alignment: Word=Every 4,Half=Every 2B

Data Transfers



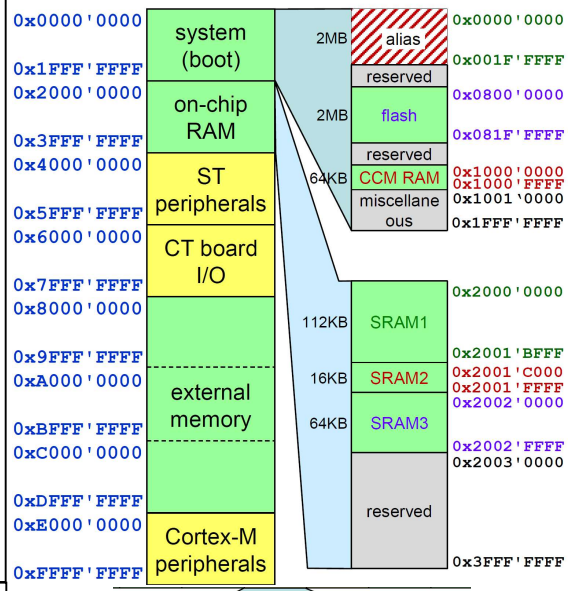
Implicit Casts(Integer Promotion=>Rank)

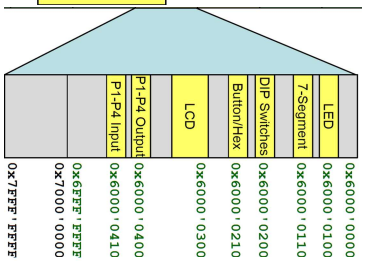
Same size=>Unsigned, Different Size=>Signed if larger and can hold all values of smaller unsigned

Expression	Type	Evaluation
0 == 0U	unsigned	1
-1 < 0	signed	1
-1 < 0U	unsigned	0
2'147'483'647 > -2'147'483'647 - 1	signed	1
2'147'483'647U > -2'147'483'647 – 1	unsigned	0
2'147'483'647 > (int) 2'147'483'648U	signed	1
-1 > -2	signed	1
(unsigned) -1 > -2	unsigned	1

Memory Map

Address space for CT board is 4 GB, each block 512 MB
SRAM: Used for stack, heap, and global variables.
Flash: Nonvolatile slower memory
CCM Ram: Core coupled very fast RAM
Alias: special memory mapping that lets you access individual bits as if they were 32-bit words, mainly for atomic bit operations.



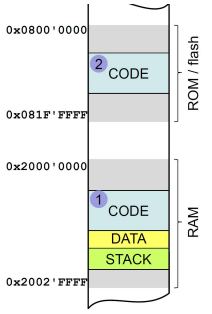


C-Type – unsigned integers	Size	Term	inttypes.h / stdint.h
unsigned char	8 Bit	Byte	uint8_t
unsigned short	16 Bit	Half-word	uint16_t
unsigned int	32 Bit	Word	uint32_t
unsigned long	32 Bit	Word	uint32_t
unsigned long long	64 Bit	Double-word	uint64_t
C-Type – signed integers	**Size**	**Term**	**inttypes.h / stdint.h**
signed char	8 Bit	Byte	int8_t
short	16 Bit	Half-word	int16_t
int	32 Bit	Word	int32_t
long	32 Bit	Word	int32_t
long long	64 Bit	Double-word	int64_t

Object File Sections

- **CODE** (1) (2)
  - Read-only → RAM or ROM
  - Instructions (opcodes)
  - Literals [1]
- **DATA** [2]
  - Read-write → RAM
  - Global variables
  - *static* variables in C
  - Heap in C → `malloc()`
- **STACK**
  - Read-write → RAM
  - Function calls / parameter passing
  - Local variables and local constants

[1] Literal: a fixed/constant value in source code




```
uint32_t  g_init_var = 0x4D5E6F70;
uint32_t  g_noinit_var;

const uint32_t g_const = 0xDDEEFF00;
void show_variables(void)
{
    uint32_t local_var;
    const uint32_t local_const =
                          0x7261504F;
    static uint32_t static_local_var;
    local_var = 0xD4E5F607;
    ...
}
```

stored in registers [1]

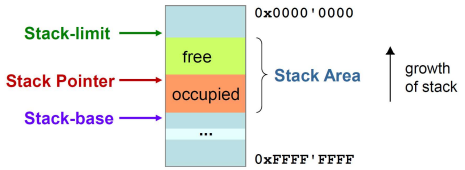
literal

```
AREA GlobalVars, DATA, READWRITE
g_init_var        DCD    0x4d5e6f70
g_noinit_var      DCD    0x00000000
static_local_var  DCD    0x00000000

AREA MyConsts, DATA, READONLY
g_const           DCD    0xddeeff00

AREA MyCode, CODE, READONLY,
show_variables    ...
                  ...
                  BX     lr
lit1              DCD    0x7261504f
lit2              DCD    0xd4e5f607
```

[1] if possible. E.g. if you use the address operator (&) on a local variable it will be allocated on the stack.

Stack

- **Stack Area (Section)** — Continuous area of RAM
- **Stack Pointer (SP)** — R13 → points to last written data value
- **PUSH { ... }** — Decrement SP and store word(s)
- **POP { ... }** — Read word(s) and increment SP
- Direction on ARM — "grows" from higher towards lower addresses → full-descending stack
- Alignment — Stack operations are **word-aligned**



Integer values based on word size

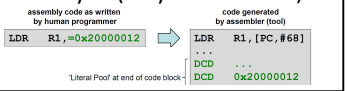
8-bit	hex	unsigned	signed
	0x00	0	0
	...	...	...
	0x7F	127	127
	0x80	128	-128
	...	...	...
	0xFF	255	-1

16-bit	hex	unsigned	signed
	0x0000	0	0
	...	...	...
	0x7FFF	32'767	32'767
	0x8000	32'768	-32'768
	...	...	...
	0xFFFF	65'535	-1

32-bit	hex	unsigned	signed
	0x0000 0000	0	0
	...	...	...
	0x7FFF'FFFF	2'147'483'647	2'147'483'647
	0x8000'0000	2'147'483'648	-2'147'483'648
	...	...	...
	0xFFFF'FFFF	4'294'967'295	-1

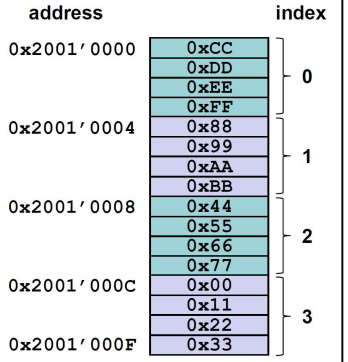
Pseudo Instruction

Shortcut that assembler translates. **LDR R0, =0x12345678**
Assembler creates literal pool in code section, typically at the end. EQU values with =CONST or #for MOVS. Load DCD values with myLit (=myLit loads add)



Arrays in Assembly

```
byte_array
        DCB     0xAA,0xBB,0xCC,0xDD
        DCB     0xEE,0xFF

word_array  DCD     0xFFEEDDCC
            DCD     0xBBAA9988
            DCD     0x77665544
            DCD     0x33221100
```



element address = base address + element size • index

Integer casting in C

Only interpretation changes

Extension: 4 Bit → 8 Bit

- Unsigned → Zero Extension

1011 → 0000 1011 0011 → 0000 0011

- Signed → Sign Extension

1011 → 1111 1011 0011 → 0000 0011

Truncation: Reduce number of digits

- Cast cuts the left most digits

Unsigned → modulo Operation

```
uint32_t x = 287962;
uint16_t sx = (uint16_t) x;
uint32_t y = (uint32_t) sx;
```

0x000464DA → 287'962
0x64DA → 25'818
0x000064DA → 25'818

Signed → possible change of sign!

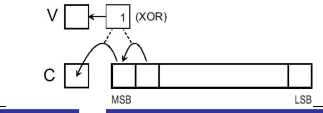
```
int32_t x  = 53191;
int16_t sx = (int16_t) x;
int32_t y  = (int32_t) sx;
```

0x0000CFC7 → 53'191
0xCFC7 → -12'345
0xFFFFCFC7 → -12'345

Flags & Conditional Branches

Flag	Condition
(**N**) Negative	MSB == 1
(**Z**) Zero	All bits == 0
(**C**) Carry	- add: carry out = 1 - sub: no borrow = 1
(**V**) Overflow	- **add**: (op1[31] == op2[31]) && (res[31] != op1[31]) - **sub**: (op1[31] != op2[31]) && (res[31] != op1[31])

Overflow



cond	short	Flag
0000	EQ	Z == 1
0001	NE	Z == 0
0010	CS/HS	C == 1
0011	CC/LO	C == 0
0100	MI	N == 1
0101	PL	N == 0
0110	VS	V == 1
0111	VC	V == 0

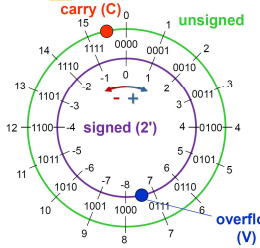
cond	short	Flag
1000	HI	C == 1 and Z == 0
1001	LS	C == 0 or Z == 1
1010	GE	N == V
1011	LT	N !== V
1100	GT	Z == 0 and N == V
1101	LE	Z == 1 or N != V
1110	AL	always
1111	--	--

For signed(Arithmetic): MI, PL, VS, VC, GE, LT, GT, LE

Negative Numbers (A – B = A + TC(B))

signed $-b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$

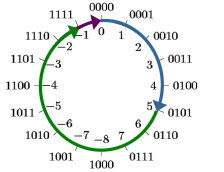
unsigned $+b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$



binary	unsigned	signed 2'-Compl.
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

■ **2' Complement**

$a - a = 0 \leftrightarrow a + OC(a) + 1 = 0$

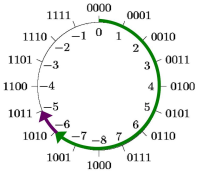


OC(a) is the bit-wise inverse of a

```
1111
- 0101
------
1010
```

OC(0101) = 1010

$-a = OC(a) + 1 = TC(a)$



-5d = TC(0101) = 1010 + 1 = 1011

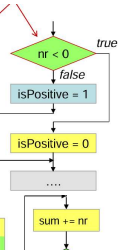
OC: 1' complement
TC: 2' complement

Control Structures

```
int32_t nr;
int32_t isPositive;
...
if (nr >= 0) {
    isPositive = 1;
}
else {
    isPositive = 0;
}
```

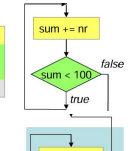
Assume: nr in R1, isPositive in R2

```
    CMP   R1,#0x00
    BLT   else
    MOVS  R2,#1
    B     end_if
else
    MOVS  R2,#0
end_if
    ....
```



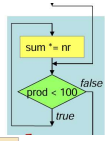

```
...
sum = 0;
do {
    sum += nr;
} while (sum < 100);
```

```
    MOVS  R2,#0
loop ADDS  R2,R2,R1
    CMP   R2,#100
    BLT   loop
    ....
```




```
...
prod = 1;
while (prod < 100) {
    prod *= nr;
}
```

```
    MOVS  R2,#1
    B     test
loop
    MULS  R2,R1,R2
test
    CMP   R2,#100
    BLT   loop
    ...
```




```
for (init-expr; test-expr; update-expr)
    body-block
```

```
init-expr;
while (test-expr) {
    body-block
    update-expr;
}
```

Jump Table

```
uint32_t result, n;
switch (n) {
case 0:
    result += 17;
    break;
case 1:
    result += 13;
    //fall through
case 3: case 5:
    result += 37;
    break;
default:
    result = 0;
}
```

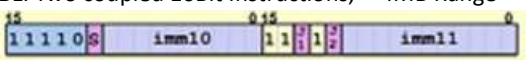
Assume: n in R1, result in R2

```
NR_CASES      EQU   6
case_switch   CMP   R1, #NR_CASES
              BHS   case_default
              LSLS  R1, #2    ; * 4
              LDR   R7, =jump_table
              LDR   R7, [R7, R1]
              BX    R7
case_0        ADDS  R2, R2, #17
              B     end_sw_case
case_1        ADDS  R2, R2, #13
case_3_5      ADDS  R2, R2, #37
              B     end_sw_case
case_default  MOVS  R2,#0
end_sw_case   ...

jump_table    DCD   case_0
              DCD   case_1
              DCD   case_default
              DCD   case_3_5
              DCD   case_default
              DCD   case_3_5
```

BC: Range for 8bit signed address from PC=>−256 to +254 Bytes of space because each instr.=>2 Byte

B: −2048 to +2046 Bytes, same reason as above

BL: Two coupled 16Bit Instructions, +-4MB Range




```
I1 = NOT(J1 EOR S); I2 = NOT (J2 EOR S)
<imm> = S:I1:I2:imm10:imm11:0
LR = PC (LSB set to '1')
PC = PC + <imm>
```

B(L)X: Jump to register address=>32bit address range=>Absolute(indirect(address in mem(reg) not in instruction) jump=>Not "movable"

Memory Sections / Endian

Lowest add + number of Bytes -1 = highest addr.

Little: LSB lower address **Big**: LSB higher addr.

Board uses Little, but written human code is Big

From C to executable

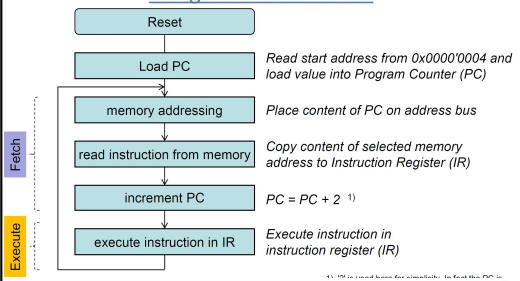
Preprocessor: .c=>.i Handles #instr., copies .h into source, removes comments, handles macros

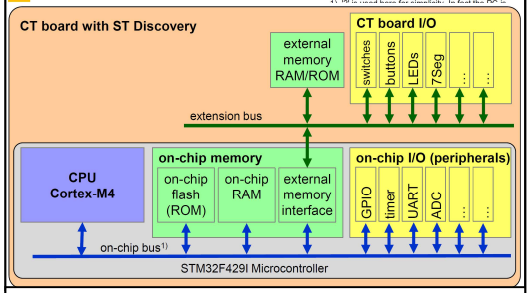
Compiler: .i=>.s Translates into Assembly, generates pseudo instructions, optimizations like Register Allocation, Loop unrolling, Constant Folding

Assembler: .s=>.o Turns into Binary File, Instruction Selection, Creates Literal pool (for pseudo)

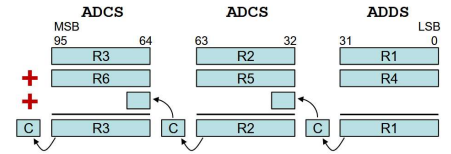
Linker: all .o=> single .axf, calculates final absolute addresses(before blank), Remove unused

Program Execution





Multi-Word Addition




```
; Operand A:        96-bit in R3(MSW),R2,R1(LSW)
; Operand B:        96-bit in R6(MSW),R5,R4(LSW)
; Result = A + B:   96-bit in R3(MSW),R2,R1(LSW)
00000028 1909     ADDS    R1,R1,R4
0000002A 416A     ADCS    R2,R2,R5
0000002C 4173     ADCS    R3,R3,R6
```

Conversion

31-28, 27-24, 23-20, 19-16

15-12, 11-8, 7-4, 3-0

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

Multiplication

Upper 32-bit are lost. For bigger: Separate A,B into hi and lo parts. A*B P1=lo*lo;P2=hi*lo;P3lo*hi;P4=hi*hi P2&3: Split into lower and upper halves: Lower P<<16, Upper P>>16

After splitting up: **ADDS** P1+Lower **ADCS** P4+Upper, Then Result in P1&4

unsigned	signed
0101 * 1101	0101 * 1101
00001101	11111101
00000000	00000000
00001101	11111101
00000000	00000000
000001000001	100111110001

zero extension of multiplier

sign extension of multiplier

Interpretation unsigned
5d * 13d = 65d → correct

Interpretation unsigned
5d * 13d = 241d → wrong

Interpretation signed
5d * – 3d = 65d → wrong

Interpretation signed
5d * – 3d = – 15d → correct

PC-relative addressing

Executing: Current Instruction X **Decoding**: Next X+2, invisible reg **Fetching**: X+4 ← PC points to this

Alignment Rule: clears last 2 bits, before offset

Subroutines

Stack operations

Order: It always writes & reads data from lowest addr. to lowest reg

POP

- **registers**
  - One or more registers to be restored
  - Low registers
    - → `reg_list` = one bit per register
  - PC (R15) → P-bit
    - → No other high registers
  - Lowest register reloaded first

```
POP {registers}
```


```
addr = SP

for i = 0 to 7
    if reg_list<i> == '1' then
        R[i] = Mem[addr,4]
        addr = addr + 4

if (P == '1') then
     PC = Mem[addr]

SP = SP + 4*BitCount(P::reg_list)
```

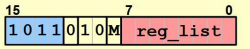
```
00000000 BC80      POP     {R7}
00000002 BC3A      POP     {R1,R3,R4,R5}
00000004 BC3A      POP     {R1,R3-R5}
00000006 BD00      POP     {PC}
00000008 BD80      POP     {R7,PC}
```

`P::reg_list = 0x03A = 0'0011'1010b`

PUSH

- **registers**
  - One or more registers to be stored
  - Low registers
    - → `reg_list` = one bit per register
  - LR (R14) → M-bit
    - → No other high registers
  - Lowest register stored first (lowest adress)

```
PUSH {registers}
```




```
addr = SP - 4*BitCount(M::reg_list)

for i = 0 to 7
    if reg_list<i> == '1' then
        Mem[addr,4] = R[i]
        addr = addr + 4

if (M == '1') then
     Mem[addr] = LR

SP = SP - 4*BitCount(M::reg_list)
```

```
00000000 B480      PUSH    {R7}
00000002 B43A      PUSH    {R1,R3,R4,R5}
00000004 B43A      PUSH    {R1,R3-R5}
00000006 B500      PUSH    {LR}
00000008 B580      PUSH    {R7,LR}
```

`M::reg_list = 0x03A = 0'0011'1010b`

ADD (SP plus immediate)

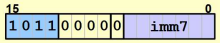
load stack pointer plus offset into register

```
ADD <Rd>,SP,#<imm>
```


```
<imm> = imm8:00
Rd = SP + <imm>
```

allocate (SUB) / deallocate (ADD) memory on stack

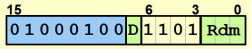
```
ADD SP,SP,#<imm>
```




```
<imm> = imm7:00
SP = SP + <imm>
```

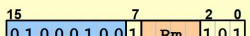
ADD (SP plus register)

```
ADD <Rdm>,SP,<Rdm>
```




```
Rdm = SP + Rdm
```

```
ADD SP,SP,<Rm>
```




```
SP = SP + Rm
```

Stack base: Initial Main Stack Pointer (MSP) value, loaded via HW before execution starts, located at Vector Table entry 0x00000000, then reads 0x00000004 => loads into PC. Above garbage or reserved space => Hard fault=>PUSH the current state onto the stack. If your SP is already "broken" => Death Loop. Stack-limit < SP < stack-base

Terminology

SubRoutine: Fragment to which control can the transferred. Routine caller=>Subroutine callee

Procedure returns no result **Function** does

Structure: Label with name, returns with BX LR

ARM Procedure Call Standard

How subroutines can be separately written, separately compiled, and separately assembled to work together => contract between a calling routine (caller) and a called routine (callee)

Layout of data	• Size, alignment, layout of fundamental data types
Register Usage	• What are the registers used for
Memory Sections and Stack	• Code, read-only data, read-write data, stack, heap
Stack	• Full-descending, word-aligned, …
Subroutine Calls	• Mechanism using LR and PC
Result Return	• Returning arguments through r0 (and r1 – r3)
Parameter Passing	• Passing arguments in r0-r3 and on stack

Register	Synonym	Role
r0	a1	Argument / result / scratch register 1
r1	a2	Argument / result / scratch register 2
r2	a3	Argument / scratch register 3
r3	a4	Argument / scratch register 4
r4	v1	Variable register 1
r5	v2	Variable register 2
r6	v3	Variable register 3
r7	v4	Variable register 4
r8	v5	Variable register 5
r9	v6	Variable register 6
r10	v7	Variable register 7
r11	v8	Variable register 8
r12	IP	Intra-Procedure-call scratch register[1)]

Register contents might be modified by callee

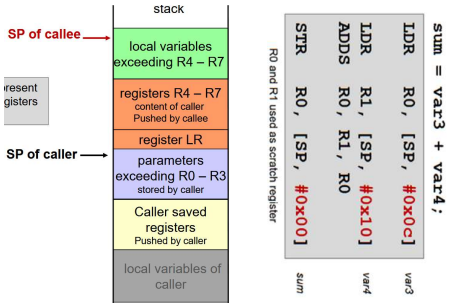
Callee must preserve contents of these registers (Callee saved)

Cortex-M0: Registers r8 – r11 have limited set of instructions. Therefore, they are often not used by compilers.

Scratch: Hold intermediate value during calculation, usually not named, limited lifetime

Variable: local to a routine, and often named in the source code, often R8-11

Return: Word or smaller R0, Double-Word R0/R1,128-bit R0-R3 with LSW stored lower

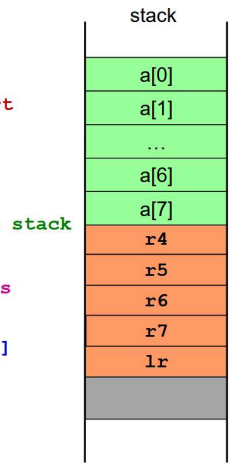


C Functions - Stack Frame

- **Example**

```
int main(void)
{
  uint32_t start = 0x1234;
  uint32_t result = 0;

  result = calc(start);
}

uint32_t calc(uint32_t val)
{
  uint32_t a[8];
  uint32_t res = 0;
  ...
  a[0] = val;
  ...
  return res;
}
```

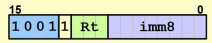
```
MOV r0,... ;start
BL  calc

PUSH {r4-r7,lr}
;allocate a[] on stack
SUB sp,sp,#0x20

MOVS  r6,#0   ;res

;access a[0]
STR r0,[sp,#0x00]

MOV  r0,r6   ;res
;stack teardown
ADD sp,sp,#0x20
POP {r4-r7,pc}
```



Stack Memory Operation

LDR (immediate) T2

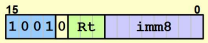
```
LDR <Rt>,[SP,#<imm>]
```




```
<imm> = imm8:00
Rt = Mem[SP + <imm>]
```

STR (immediate) T2

```
STR <Rt>,[SP,#<imm>]
```




```
<imm> = imm8:00
Mem[SP + <imm>] = Rt
```

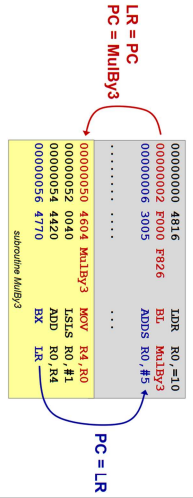
PUSH {R2,R3,R6}

```
00000000 B083      SUB     SP,SP,#12
00000002 9200      STR     R2,[SP]
00000004 9301      STR     R3,[SP,#4]
00000006 9602      STR     R6,[SP,#8]
```

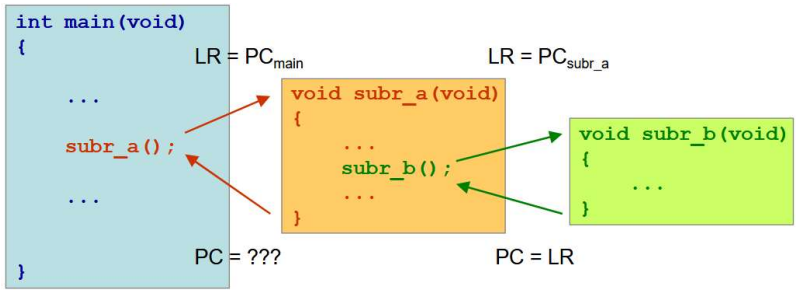
POP {R2,R3,R6}

```
00000008 9A00      LDR     R2,[SP]
0000000A 9B01      LDR     R3,[SP,#4]
0000000C 9B02      LDR     R6,[SP,#8]
0000000E B003      ADD     SP,SP,#12
```

Control flow



Nested



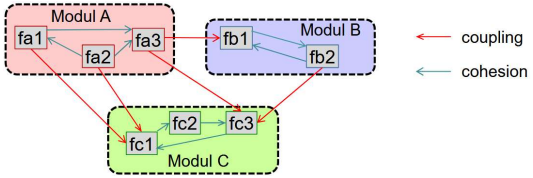
Modular Programming

Complexity & Guidelines

Topic	Benefits
Enable working in teams	Multiple developers working on the same source repository
Useful partitioning and structuring of the programs	Eases reusing of modules
Individual verification of each module	Benefits all users of the module
Providing libraries of types and functions	For reuse instead of reinvention
Mixing of modules that are programmed in various languages	E.g. mix C and assembly language modules
Only compile the changed modules	Speeds up compilation time

High Cohesion: Group together what belongs => each module single task, lean external interface

Low Coupling: Little dependencies=>split up



Divide and conquer: Partition functionality into manageable chunks => Hierarchical design

Information hiding: Split interface from implementation=>freedom, dont disclose unnecessary details

Module design and implementation

Module Interface: .h defines what functionality is available to the client of the code

Module Implementation: .c Implement Interface

Testing & Usage: Modules can be tested individually IF designed correctly => they should be designed to be reused from the start

Partitioning: Modular Programming(Source code base is split into multiple files), Each source file defines a module=>gets translated into one object file

C Implications: C declarations and definitions, Header files to share commonly used declarations

Challenge: Require concept where Types, functions and variables may be defined in other modules than where they are used, Consistency of types, functions and variables is maintained across module boundaries **Solution** Declared-before-used, One-definition-rule

C: Declaration vs. definition

Declaration: Specifies how a name can be used

```
uint32_t square(uint32_t v); // square function defined elsewhere
extern uint32_t counter;     // counter variable defined elsewhere
struct S;                    // struct S type defined elsewhere
```

Definition: Where a function is given with its body, where memory is allocated for a variable, A struct type with its members

```
uint32_t square(uint32_t v) { ... } // square function definition
uint32_t counter;                   // counter variable definition
struct S { ... };                   // struct S type definition
```

Names declared before use: Each name must be declared before it can be used, a definition is also a declaration=> May leads to repeating declarations

```
// program_A.c

// declaration of square
uint32_t square(uint32_t v);
...
int main(void) {
  // use of square
  res = square(a) + b;
  ...
}
```

```
// program_B.c

// declaration of square
uint32_t square(uint32_t v);
...
int main(void) {
  // use of square
  y = square(x);
  ...
}
```

One-definition-rule: A variable or function may be declared multiple times, but only once in the same scope

Header Files: Use a single header file instead of duplicating declarations =>Maintains consistency over time

```
// square.h
#ifndef _SQUARE_H_ // incl.-
#define _SQUARE_H_ // guard

// declaration of square
uint32_t square(uint32_t v);

#endif // end of incl.-guard
```

```
// square.c
#include "square.h"

// definition of square
uint32_t square(uint32_t v)
{
  return v*v;
}
```

Usage through **`#include`** preprocessor directive

```
// program_A.c
#include "square.h"
int main(void) {
  res = square(a) + b;
  ...
}
```

```
// program_B.c
#include "square.h"
int main(void) {
  y = square(x);
  ...
}
```

R

Linkage

The global name can be **externally** or **internally** available for use in any modules => function or variable => If neither => No Linkage

All global names have external linkage unless defined **`static`**

```
// square.c
...
uint32_t square(uint32_t v) {
  return v*v;
}
```

```
// main.c
#include "square.h"
static uint32_t a = 5;
static uint32_t b = 7;
int main(void) {
  uint32_t res;
  res = square(a) + b;
  ...
}
```

square = external linkage

a	= internal linkage
b	= internal linkage
main	= external linkage
res	= no linkage
square	= external linkage[1)]

From C declaration/definition to assembly

- Names given in C translate into symbols in assembly
- C-definitions with external linkage translate into EXPORT symbols in assembly
- C-declarations with external linkage which are used but not defined in the module translate into IMPORT symbols in assembly

```
// square.c
...
uint32_t square(uint32_t v) {
  return v*v;
}
```

→

```
; square.s
    AREA myCode,CODE,READONLY
    EXPORT square
square PROC
    MOV     r1,r0
    MULS    r0,r1,r0
    BX      lr
    ENDP
    END
```

Linkage control

- EXPORT declares a symbol for use by other modules
- IMPORT declares a symbol from another module for use in this module

Internal symbols

- Neither EXPORT nor IMPORT
- Defined in this module
- Can only be used within this module

```
// main.c
#include "square.h"
static uint32_t a = 5;
static uint32_t b = 7;
int main(void) {
  uint32_t res;
  res = square(a) + b;
  ...
}
```

internal symbols

```
; main.s                                   usable outside of module main
    AREA myCode,CODE,READONLY
    EXPORT main
    IMPORT square                          from module square
main PROC
    LDR     r0,a_adr
    LDR     r0,[r0,#0]   ; a
    BL      square
    ...
    ENDP
a_adr DCD   a
b_adr DCD   b

    AREA myData, DATA
a   DCD     0x00000005
b   DCD     0x00000007
```

Linker

From assembly symbols to object file symbols

References: Imported symbols from assembly code translate to global reference symbols in the object file

Global: Exported symbols from assembly code translate to global symbols in the object file

Local: Internal symbols from assembly code translate to local symbols in the object file

Exported
- Code symbol `square`

Referenced/Imported
- None
- No external symbol used

Local
- None
- No internal symbol defined

```
; square.s
    AREA myCode,CODE,READONLY
    EXPORT square
square PROC
    MOV     r1,r0
    MULS    r0,r1,r0
    BX      lr
    ENDP
    END
```

Assemble

Ref: -
Global: square (code)
Local: -
square.o

```
// square.c
uint32_t square(uint32_t v)
{
  return v*v;
}
```

```
// main.c
uint32_t square(uint32_t v);
static uint32_t a = 5;
static uint32_t b = 7;
int main(void) {
  uint32_t res; //stack or reg
  res = square(a) + b;
}
```

Preprocess, compile and assemble

Ref.: -
Global: square (code)
Local: -
square.o

Ref.: square (code)
Global: main (code)
Local: a, b (data)
main.o

Object Files

main.o (part III)

- File section #7: relocation table:
  - Relocation at code address `0x00000006`:
    - ► Modify the BL call to branch to the symbol `square`
  - Relocation at code address `0x00000014`:
    - ► Set the absolute 32 bit value of the symbol `a`
  - Relocation at code address `0x00000018`:
    - ► Set the absolute 32 bit value of the symbol `b`

Relocation table section

```
...
** Section #7 '.rel.text' (SHT_REL)
 #       Offset    Relocation Type              Wrt Symbol
=============================================================
 0  0x00000006   10 R_ARM_THM_CALL            12 square
 1  0x00000014    2 R_ARM_ABS32                7 a
 2  0x00000018    2 R_ARM_ABS32                8 b
...
```

Affected code section locations

```
** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
...
0x00000006:    f7fffffe    ....    BL      square      ; BL needs adjustment
...
0x00000014:    00000000    ....    DCD     0 ; address a will be stored here
0x00000018:    00000000    ....    DCD     0 ; address b will be stored here
```

Linker Tasks

Merge: object files data sections(place all data sections of the individual object files into one data section of the executable file), object file code sections (Place all code sections of the individual object files into one code section of the exec file)

Resolve: used external symbols=> Search missing addresses of used external symbols

symbols (main.o)

```
 #  Symbol Name   Value      Bind  Sec  Type  Vis  Size
=================================================================
 7  a             0x00000000 Lc    4    Data  De   0x4
 8  b             0x00000004 Lc    4    Data  De   0x4
11  main          0x00000001 Gb    1    Code  Hi   0x14
12  square        0x00000000 Gb  Ref    Code  Hi
```

symbols (square.o)

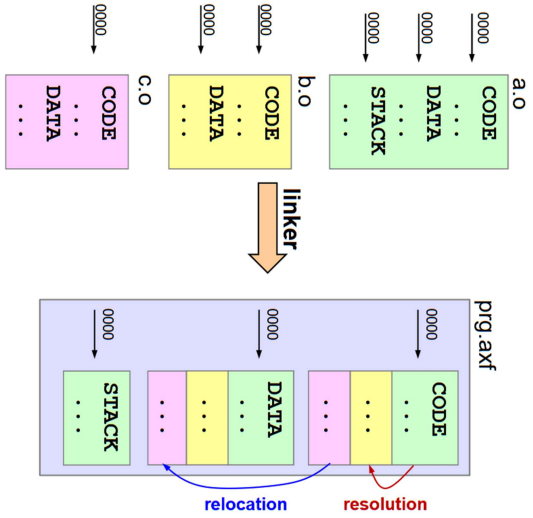
```
 #  Symbol Name   Value      Bind  Sec  Type  Vis  Size
=================================================================
 6  square        0x00000001 Gb    1    Code  Hi   0x8
```

resolved symbols

```
 #  Symbol Name   Value      Bind  Sec  Type  Vis  Size
=================================================================
20  a             0x00000000 Lc    4    Data  De   0x4    (main.o)
21  b             0x00000004 Lc    4    Data  De   0x4    (main.o)
186 main          0x00000001 Gb    1    Code  Hi   0x14   (main.o)
187 square        0x00000000 Gb    1    Code  Hi   0x8    (square.o)
```

Relocate addresses: Adjust used addresses since merging the sections invalidated the original addresses. The linker places the sections depending on the target system, the given command line arguments and the scatter file (if given). new value = global base + merge offset + module relative offset

global base	= internal SRAM	= 0x20000000
merge offset	= 1st in merged data section	= 0x00000000
module relative offset	= b is the 2nd variable after a	= 0x00000004
new value for symbol b		= 0x20000004



Executable File(All Linked)

If the program is loaded before execution (by a loader of the hosting operating system), there might still be **1.** Unresolved symbols for linking with shared (dynamic linked) libraries **2.** A relocation table to move the program/data to fixed locations **AXF** = ARM eXecutable

Code section: Code and constant data of the program

Data section: global vars of the program

Symbol table: All symbols with their attributes like global/local, etc.

Tool chain

Tool chain: View (Set of tools that is required to create from source code an exec for given env), Native (Builds for same architecture as it runs), Cross compilation (Another architecture, e.g. Keil)

Libraries: Collection of object files, May speed up linking(prepared sorted symbol table), may smaller code(only really needed parts), interchangeable, Static (Executable is completely linked with a static library at link time => self-contained), Dynamic/Shared (Needs other libraries at run time=>smaller exec, version!!)

Debugging: Single stepping(HW or SW supported), Source level needs mapping between machine addr and mem location, code lines/type

Object Files

Contain all compiled data of module

Code section: Code and constant data of the module, based at address 0x0

Data section: All global variables of the module, based at address 0x0

Symbol table: All symbols with their attributes like global/local, reference, etc.

Relocation table: Which bytes of the data and code section need to be adjusted (and how) after merging the sections in the linking process

Executable and Linkable Format: ELF is used by ARM tool chain, includes all above and additional

File section #1: code section, at base address `0x00000000`

File section #5: symbol table: `square` = `global code symbol`

No data section (has no global variables)

No relocation section (no referenced symbols in code/data)

```
File Type: ET_REL (Relocatable object) (1)
...
** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Address: 0x00000000
square
        0x00000000:    4601        .F      MOV     r1,r0
        0x00000002:    4608        .F      MOV     r0,r1
        0x00000004:    4348        HC      MULS    r0,r1,r0
        0x00000006:    4770        pG      BX      lr
...
** Section #5 '.symtab' (SHT_SYMTAB)
 #  Symbol Name                  Value      Bind  Sec  Type  Vis  Size
=================================================================
 6  square                       0x00000001 Gb    1    Code  Hi   0x8
```

square.o
0000 → CODE ...

main.o (part I)

- File section #1: code section, at base address `0x00000`
  - `0x00000002: LDR r0, =a` (address **a** stored at 0x14)
  - `0x0000000a: LDR r1, =b` (address **b** stored at 0x18)
  - `BL square` calls a dummy address until linked
- File section #4: data section, at base address `0x00000`

```
** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Address: 0x00000000
main
        0x00000000:    b510        ..      PUSH    {r4,lr}
        0x00000002:    4804        .H      LDR     r0,[pc,#16] ; LDR r0, =a
        0x00000004:    6800        .h      LDR     r0,[r0,#0]  ; r0 = value at a
        0x00000006:    f7fffffe    ....    BL      square      ; BL needs adjustment
        0x0000000a:    4903        .I      LDR     r1,[pc,#12] ; LDR r1, =b
        0x0000000c:    6809        .h      LDR     r1,[r1,#0]  ; r1 = value at b
        0x0000000e:    1844        D.      ADDS    r4,r0,r1
        0x00000010:    2000        .       MOVS    r0,#0
        0x00000012:    bd10        ..      POP     {r4,pc}
        0x00000014:    00000000    ....    DCD     0 ; address a will be stored here
        0x00000018:    00000000    ....    DCD     0 ; address b will be stored here
** Section #4 '.data' (SHT_PROGBITS) [SHF_ALLOC + SHF_WRITE]
    Address: 0x00000000
        0x00000000:    00000005                ; value at a = 5
        0x00000004:    00000007                ; value at b = 7
```

square requires esolution

Idresses of a and b require elocation

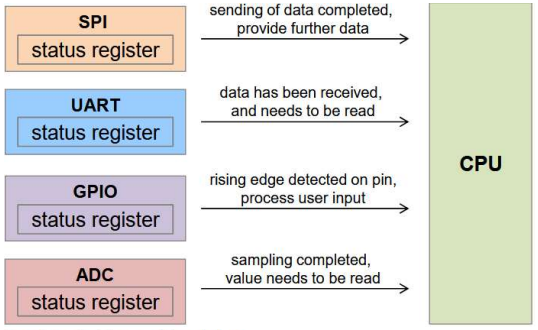
main.o (part II)

- File section #6: symbols:
  - `a`: local data section symbol, at offset 0x00000000
  - `b`: local data section symbol, at offset 0x00000004
  - `main`: global code section symbol, at offset 0x00000000 (LSB set: Thumb code)
  - `square`: global code section symbol, referenced (no definition in main.o)

```
...
** Section #6 '.symtab' (SHT_SYMTAB)
 #  Symbol Name                  Value      Bind  Sec  Type  Vis  Size
=================================================================
 7  a                            0x00000000 Lc    4    Data  De   0x4
 8  b                            0x00000004 Lc    4    Data  De   0x4
11  main                         0x00000001 Gb    1    Code  Hi   0x14
12  square                       0x00000000 Gb  Ref    Code  Hi
...
```

Control Flows

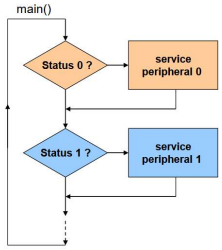
Events vs. Polling



Reading of status registers in loop
Synchronous with main program
Advantages
- Simple and straightforward
- Implicit synchronization
- Deterministic
- No additional interrupt logic required

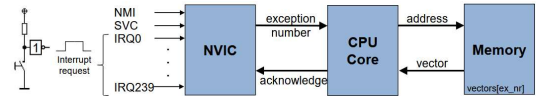
Disadvantages
- Busy wait → wastes CPU time
- Reduced throughput
- Long reaction times

in case of many I/O devices or if the CPU is working on other tasks



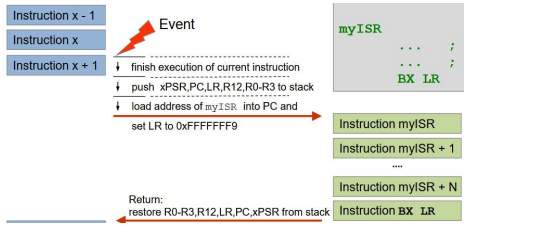
Interrupt-System

Nested Vectored Interrupt Controller (NVIC): 240 sources can trigger exception ? high level signal on IRQx, Forwards respective exception number to CPU



Vector Table: array of 32-bit addresses. These addresses are the starting points (vectors) for exception handlers and Interrupt Service Routines (ISRs)

Storing the Context: Interrupt event can take place at any time => requires saving of all registers and FLAGS=>ISR Call does this for xPSR, PC, LR, R12, R3-0. On return Load EXC_RETURN into PC(BX LR), which pops above from stack (or manually pop to PC if LR has been pushed)

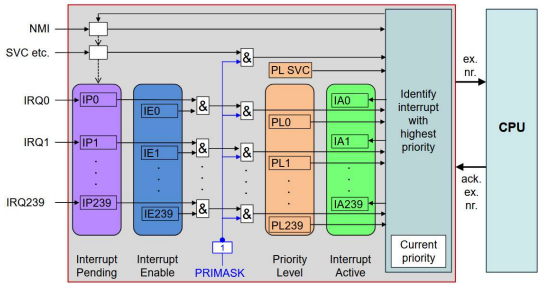


Exceptions on Cortex M3&4

System Exceptions: Reset(Restart of Processer), Non-maskable Interrupt(NMI=> Condition can't be ignored, e.g. hardware error), Faults (undefined instructions, unaligned access), System Level Calls(OS)

Exception Number	Exception Type	Priority	Description
1	Reset	−3 (Highest)	Reset
2	NMI	−2	Nonmaskable interrupt (external NMI input)
3	Hard Fault	−1	All fault conditions, if the corresponding fault handler is not enabled
4	MemManage Fault	Programmable	Memory management fault; MPU violation or access to illegal locations
5	Bus Fault	Programmable	Bus error; occurs when AHB interface receives an error response from a bus slave (also called *prefetch abort* if it is an instruction fetch or *data abort* if it is a data access)
6	Usage Fault	Programmable	Exceptions due to program error or trying to access coprocessor (the Cortex-M3 does not support a coprocessor)
7–10	Reserved	NA	–
11	SVCall	Programmable	System Service call
12	Debug Monitor	Programmable	Debug monitor (breakpoints, watchpoints, or external debug requests)
13	Reserved	NA	–
14	PendSV	Programmable	Pendable request for system device
15	SYSTICK	Programmable	System Tick Timer

Nested Vectored Interrupt Controller (NVIC)

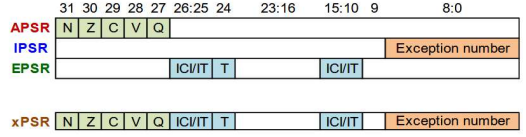


Interrupt

Sudden change of program flow due to an event => no busy wait & short reaction times but no sync between main program and IRS and difficult debug

Program Status Registers (PSRs)

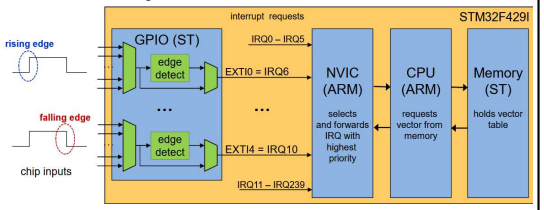
APSR Application Program Status Register
IPSR Interrupt Program Status Register
EPSR Execution Program Status Register



xPSR Combination of all three PSRs

External Interrupt Pins

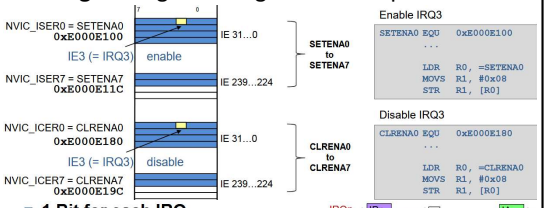
- **Chip Vendor (ST) adds GPIO logic**
  - EXTI0 through EXTI4 connected to IRQ6 through IRQ10
    - IRQ0 – IRQ5 / IRQ11 – IRQ239 used for other sources e.g. SPI, UART, ADC
  - Select chip input for each EXTIx line
  - Select level or edge



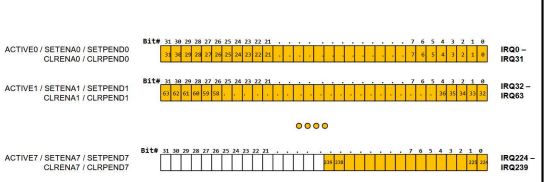
Masking of Interrupts

PRIMASK: Priority Mask Register, it uses a **single bit** to control all **maskable** interrupts. Default state is = 0 => all globally enabled=>Master switch

Masking: Telling CPU to ignore Interrupts



1 Bit for each IRQ
- Registers are written as 32-bit words
- Only bits with value "1" have an effect, "0" is ignored
- Separate registers to read and write



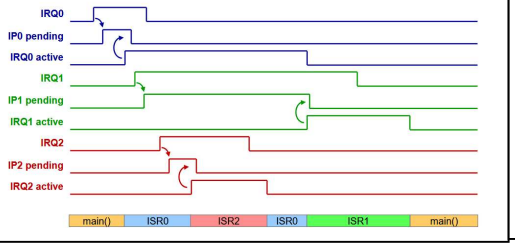
Nested Exceptions

Priority Levels define whether A can preempt(interrupt) B. Priority saved as 4bit(lower=>more)

Example Priorities

- ISR1 does not preempt ISR0
- ISR2 preempts ISR0

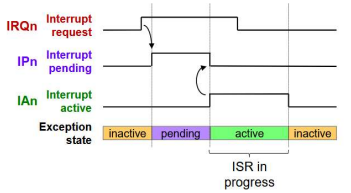
IRQ0	PL0 = 0x2	medium priority
IRQ1	PL1 = 0x3	lowest priority
IRQ2	PL2 = 0x1	highest priority



Interrupt Control

Exception States: Inactive, pending (Waiting to be served by CPU), Active(actively served by CPU), Active and Pending(From same source)

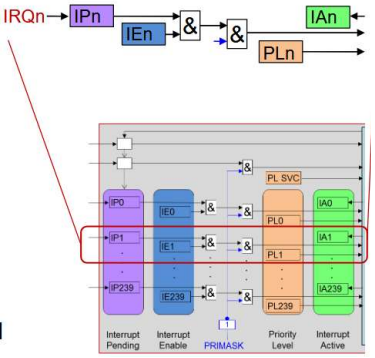
IRQ Inputs and Pending Behavior



- High level on IRQn sets pending bit (IPn)
- Active Bit (IAn)
  - Set as soon as exception is being serviced
  - Resets pending bit

[1] IRQn is set by hardware and usually reset by software

Same logic exists for each interrupt n = 0 … 239



Pending registers can be set, but active are read-only

Trigger IRQ3 by Software

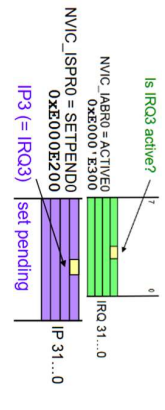
```
SETPEND0  EQU 0xE000E200
          ...
          LDR  R0,=SETPEND0
          MOVS R1,#0x08
          STR  R1,[R0]
```

Delete Pending IRQ3

```
CLRPEND0 EQU 0xE000E280
         ...
         LDR  R0,=CLRPEND0
         MOVS      R1,#0x08
         STR  R1,[R0]
```

Test if IRQ3 is active

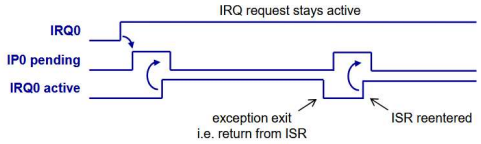
```
ACTIVE0 EQU 0xE000E300
        ...
        LDR  R0, =ACTIVE0
        MOVS R1, #0x08
        LDR  R2, [R0]
        TST  R1, R2
        BEQ  ...
```



Special Interrupt Conditions

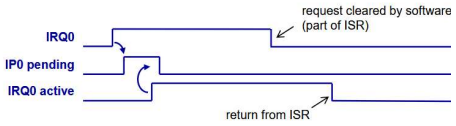
IRQ request stays active

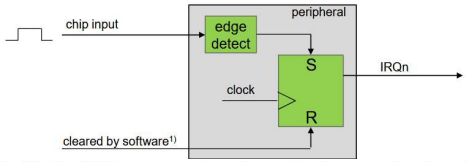
- Interrupt becomes pending again



Common Configuration on Microcontrollers

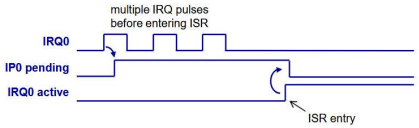
- IRQ Set by Hardware – Cleared by Software





Multiple IRQ request pulses before entering ISR

- Treated as single interrupt
- Events are lost



CMSIS: Cortex Microcontroller Software Interface Standard, Vendor-independent hardware abstraction layer for Cortex-M, defines tools for C

- **NVIC Control**

```
void NVIC_EnableIRQ(IRQn_t IRQn)              Enable IRQn
void NVIC_DisableIRQ(IRQn_t IRQn)             Disable IRQn

uint32_t NVIC_GetPendingIRQ (IRQn_t IRQn)     Return true (IRQ-Number) if IRQn is pending
void NVIC_SetPendingIRQ (IRQn_t IRQn)         Set IRQn pending
void NVIC_ClearPendingIRQ (IRQn_t IRQn)       Clear IRQn pending status
uint32_t NVIC_GetActive (IRQn_t IRQn)         Return '1' if active bit of IRQn is set. '0' otherwise

void NVIC_SetPriority (IRQn_t IRQn, uint32_t priority)  Set priority for IRQn
uint32_t NVIC_GetPriority (IRQn_t IRQn)       Read priority of IRQn

void NVIC SystemReset (void)                  Reset the system
```

Data Consistency

Data structure must not be changed during output =>Disable Interrupts during output=>Multitasking problem

```
typedef struct  {
    uint8_t minutes;
    uint8_t seconds;
} time_t;

static time_t time = { 0, 0 };

int main(void)
{
  while (1) {
    __disable_irq();
    write_byte(ADDR_LED_7_0,
                time.seconds);
    write_byte(ADDR_LED_15_8,
                time.minutes);
    __enable_irq();
  }
}
```

```
void IRQ1_Handler(void)
{
  time.seconds++;
  if (time.seconds > 59) {
    time.seconds = 0;
    time.minutes++;
    }
}
```

time = { 15, 59 }
1) Output 59 → LED_7_0
2) Interrupt → time = { 16, 0 }
3) Output 16 → LED_15_8
→ display 16 59 !!!

Disable all Interrupts

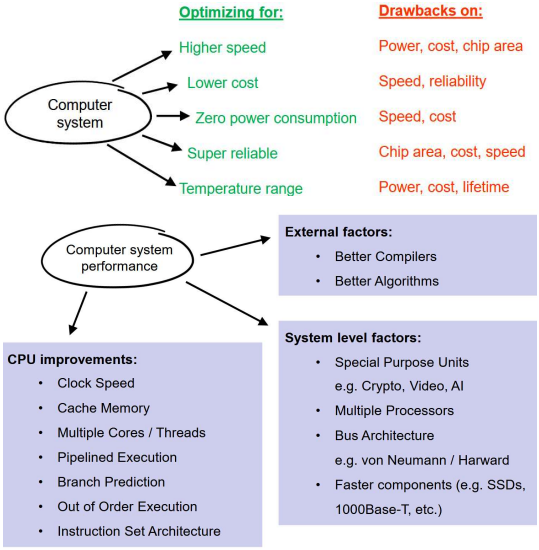
Field	Input
Assembler - Ausschalten	`cpsid i`
Assembler - Einschalten	`cpsie i`
C - Ausschalten	`__disable_irq();`
C - Einschalten	`__enable_irq();`

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

System Performance

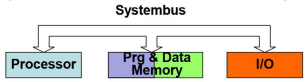
Wishlist



Bus Architecture

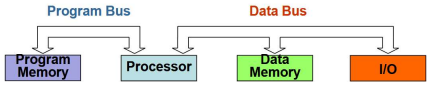
von Neumann Architecture

- Same memory holds program and data
- Single bus system between CPU and memory



Harvard Architecture

- "Mark I" at Harvard University (Howard Aiken, 1939-44)
- Separate memories for program and data
- Two sets of address/data buses between CPU and memory

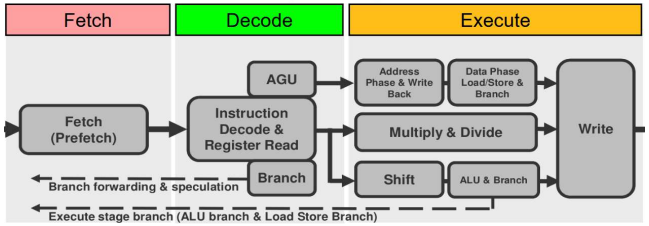


Instruction Set Architecture

RISC: Few instructions, unique instruction format, Fast decoding, simple addressing, less hardware allows higher clock rates, more chip space for registers (up to 256!), Load-store architecture reduces memory accesses, CPU works at full-speed on registers, fixed length instructions

CISC: One of the operands of an instruction may directly be a memory location, less program memory needed with complex instructions=>faster memory Modern x86 CPUs convert instruction

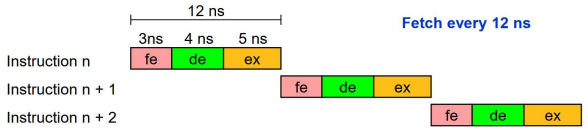
Pipelining



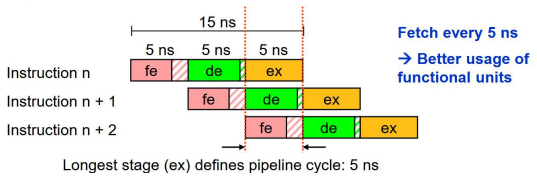
Branch Forwarding & Speculation: speculation to guess the path and minimize the penalty if it must flush the pipeline and start over. If the *Execute* stage calculates a value that the next instruction in the *Decode* stage needs immediately, "forwarding" allows the data to be passed directly backward without waiting for it to be written to memory and read back out.

Timings: Fetch (3ns), decode (4ns), execute (5ns)

Sequential execution



Pipelined execution



Instructions per second: W/O pipeline 1/Instruction delay, with pipeline 1/Max stage delay, Cortex 1/12 or 1/5

Advantages: All stages are set to the same execution time, massive performance gain, Simpler hardware at each stage allows for a higher clock rate **Disadvantages** A blocking stage blocks whole pipeline, Multiple stages may need to have access to the memory at the same time **Stall** happens if instruction takes longer like LDR or branch jump decision **Branch prediction**=> Store last taken branch **Instruction prefetch**: Fetch several at once but maybe out of order execution=> If one like LDR stalls it might already go to next one=> Complex Optimizations=> severe security problems=>Meltdown/Spectre Attack=> allow a process to access the data of another process.

Parallel Computing

Streaming/Vector Processing

- One instruction processes multiple data items simultaneously

Multithreading

- Multiple programs/threads share a single CPU

Multicore Processors

- One processor contains multiple CPU cores

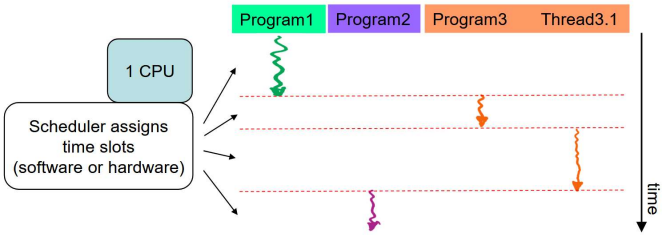
Multiprocessor Systems

- A computer system contains multiple processors

		Data streams	
		Single	Multiple
Instruction streams	Single	**SISD**: Single processor with one instruction, data stream	**SIMD**: Data Vectors, all data streams react to one instruction
	Multiple	(no examples)	**MIMD**: Multiprocessor with SIMD-instructions

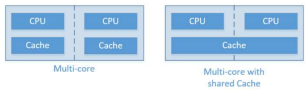
Multithreading

- Scheduler: Assigns time slots to programs/threads
- Programs/threads only **seem** to run in parallel (1 CPU)

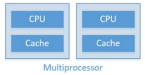


Parallelism on CPU / processor level:

- Multicore processor
  - All on one chip
  - Less traffic (cores integrated on one chip)
  - Possibility to share memories on-chip
  - Cheaper



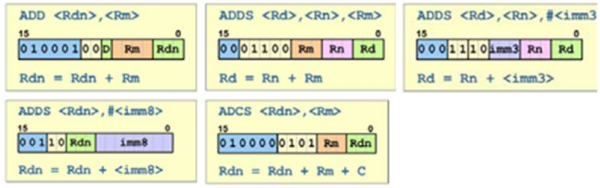
- Multiprocessor
  - Multiple Chips
  - Longer distances between CPUs
  - More expensive



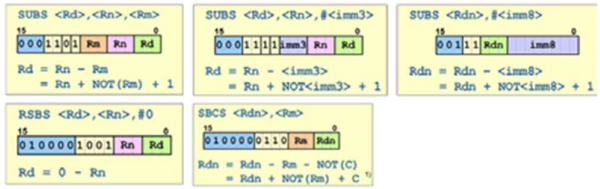
Handout

MOV

ADD

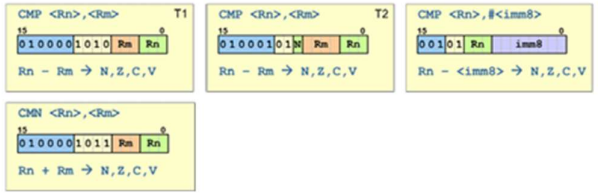


Subtract

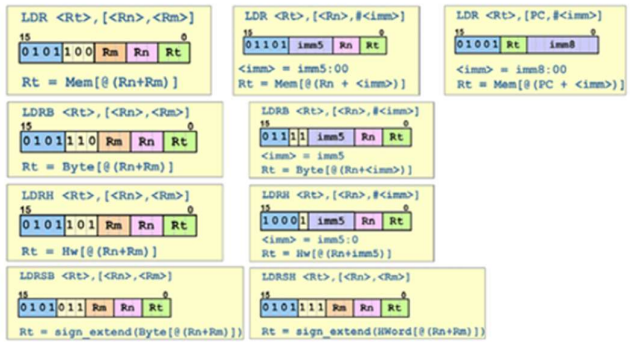


Multiply

Compare



Load



Flag- Dependent

Symbol	Condition	Flag
EQ	Equal	Z == 1
NE	Not equal	Z == 0
CS	Carry set	C == 1
CC	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

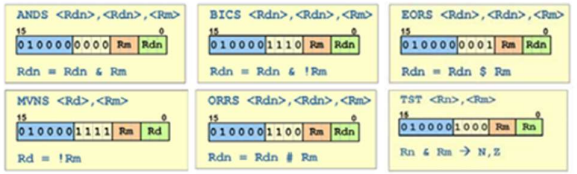
Arithmetic - unsigned: higher and lower

Symbol	Condition	Flag
EQ	Equal	Z == 1
NE	Not equal	Z == 0
HS (=CS)	Unsigned higher or same	C == 1
LO (=CC)	Unsigned lower	C == 0
HI	Unsigned higher	C == 1 and Z == 0
LS	Unsigned lower or same	C == 0 or Z == 1

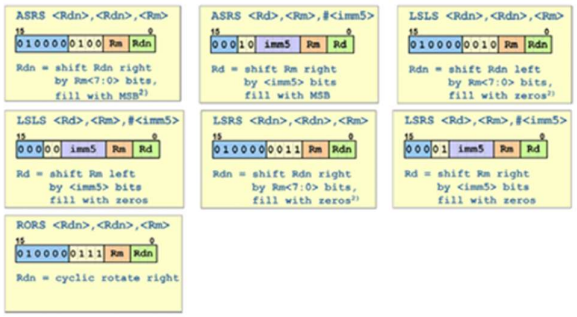
Arithmetic - signed: greater and less

Symbol	Condition	Flag
EQ	Equal	Z == 1
NE	Not equal	Z == 0
MI	Minus/negative	N == 1
PL	Plus/positive or zero	N == 0
VS	Overflow	V == 1
VC	No overflow	V == 0
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

Logical

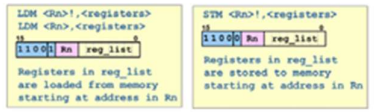


Shift/Rotate



Store

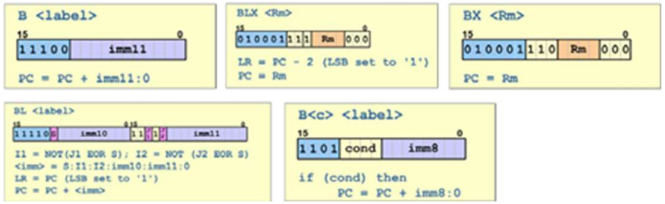
Load/Store Multiple



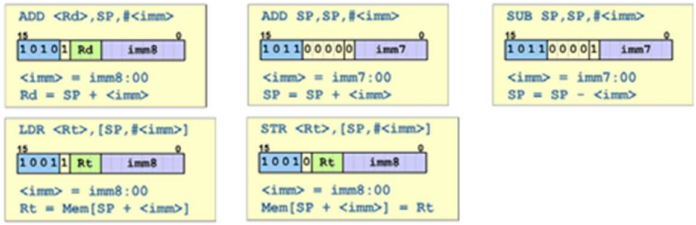
Condition Field	
Mnemonic	Description
EQ	Equal
NE	Not equal
CS / HS	Carry Set / Unsigned higher or same
CC / LO	Carry Clear / Unsigned lower
MI	Negative
PL	Positive or zero
VS	Overflow
VC	No overflow
HI	Unsigned higher
LS	Unsigned lower or same
GE	Signed greater than or equal
LT	Signed less than
GT	Signed greater than
LE	Signed less than or equal
AL	Always. Do not use in B{cond}

Push/Pop

Branch



Stack Operations



Extend