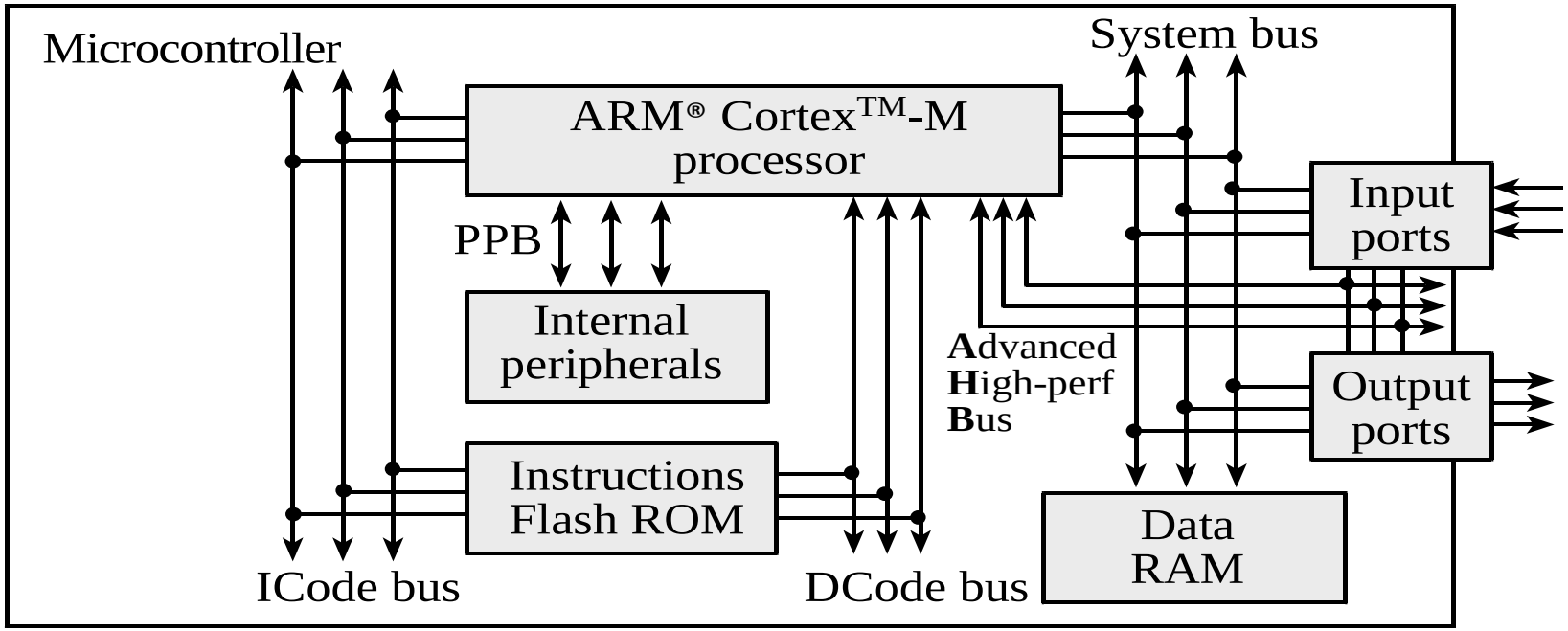
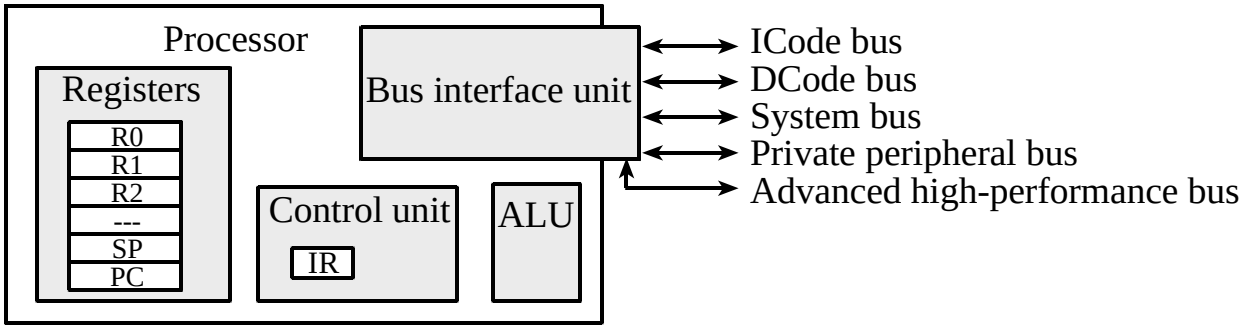
ARM cortex M4

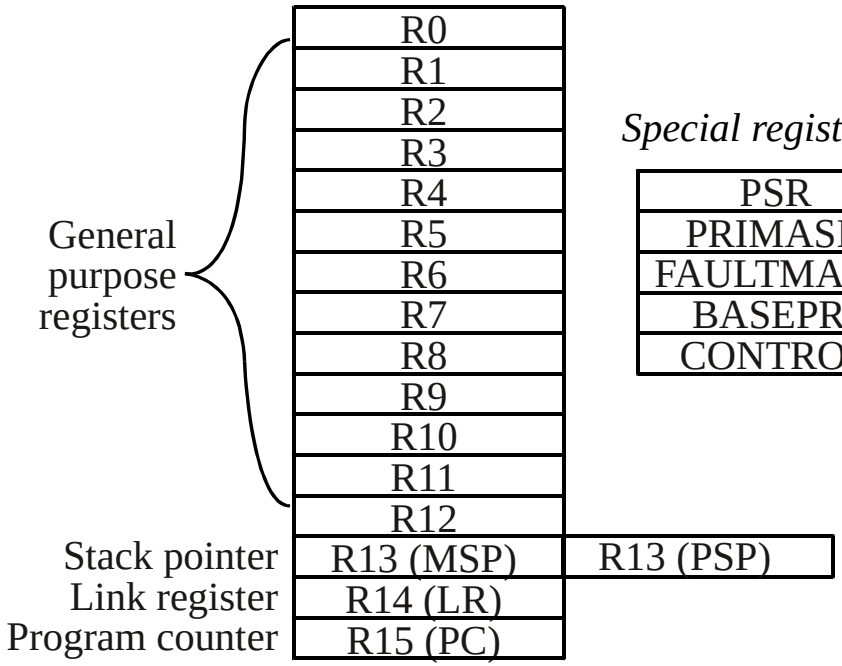
ARM Cortex M4 Block Diagram



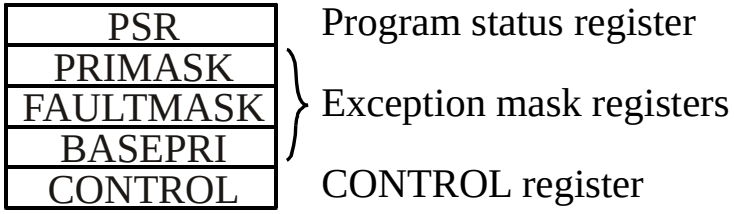
- ARM Cortex-M4 processor
- *Harvard* architecture
  - Different busses for instructions and data

Cortex M Architecture

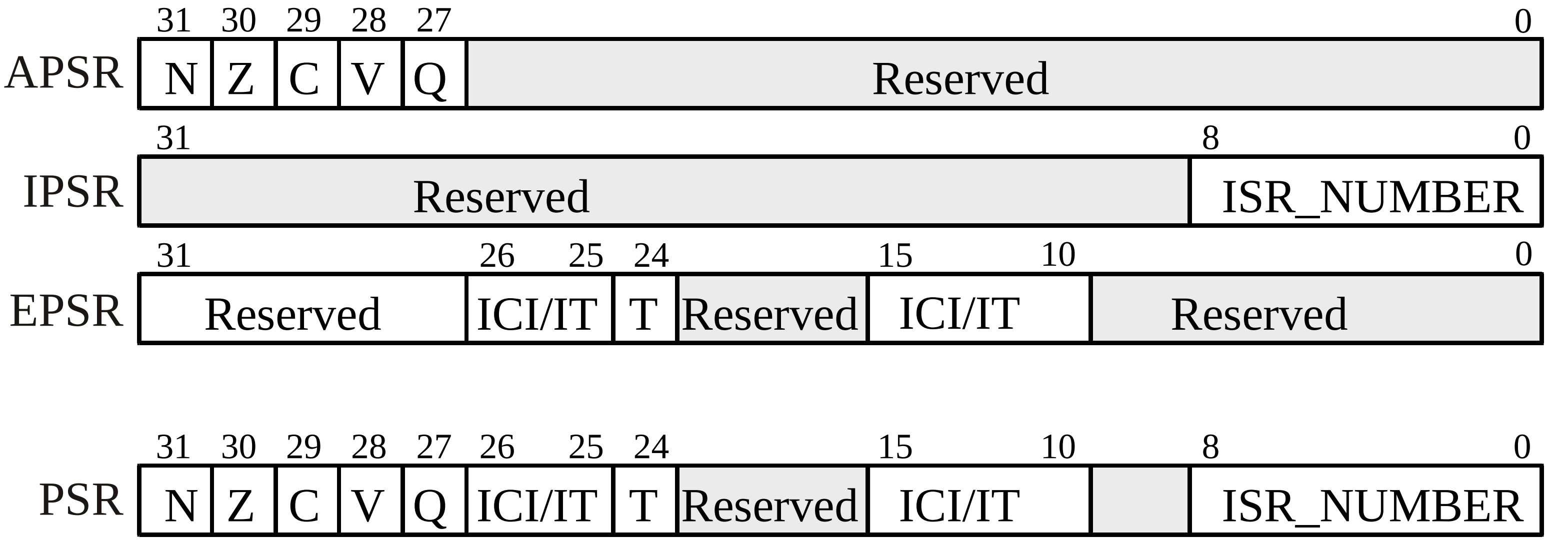




Special registers



Cortex M Architecture



	31	30	29	28	27	0
APSR	N	Z	C	V	Q	Reserved

	31	8 0
IPSR	Reserved	ISR_NUMBER

	31	26 25	24		15 10	0
EPSR	Reserved	ICI/IT	T	Reserved	ICI/IT	Reserved

	31	30	29	28	27	26 25	24		15 10		8 0
PSR	N	Z	C	V	Q	ICI/IT	T	Reserved	ICI/IT		ISR_NUMBER

APSR contains the current state of the condition flags from previous instruction executions

IPSR contains the exception type number of the current Interrupt Service Routine (ISR)

EPSR contains the Thumb state bit and the execution state bits for the If-Then (`IT`) instruction

Development of the ARM Architecture

v4	v5	v6	v7
Halfword and signed halfword / byte support System mode Thumb instruction set (v4T)	Improved interworking CLZ Saturated arithmetic DSP MAC instructions Extensions: Jazelle (5TEJ)	SIMD Instructions Multi-processing v6 Memory architecture Unaligned data support Extensions: Thumb-2 (6T2) TrustZone® (6Z) Multicore (6K) Thumb only (6-M)	Thumb-2 Architecture Profiles 7-A - Applications 7-R - Real-time 7-M - Microcontroller

- **Note that implementations of the same architecture can be different**
  - Cortex-A8 - architecture v7-A, with a 13-stage pipeline
  - Cortex-A9 - architecture v7-A, with an 8-stage pipeline

ARM Cortex M4 Instruction Set

ARM Instruction Set

Conditional execution : An instruction is only executed when a specific condition has been satisfied

Control Flow Instructions (2 instructions)

- Conditional Execution
- Branch Instructions
- Branch and Link Instructions
- Subroutine Return Instructions

Data Processing Instructions (18 instructions)

- Arithmetic
- Comparison
- Logical
- Data Movement

Load-Store Instructions (4 instructions)

- Single register transfer instruction
- Multiple register transfer instruction
- Swap instruction

ARM Instruction Set

Control Flow Instructions

Conditional Execution and Flags

- ARM instructions can be made to execute conditionally by post fixing them with the appropriate condition code field.
  - This improves code density *and* performance by reducing the number of forward branch instructions.

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

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

- By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using "S". CMP does not need "S".

```
loop
  …
  SUBS r1,r1,#1
  BNE loop
```

decrement r1 and set flags

if Z flag clear then branch

Condition Codes

- The possible condition codes are listed below
  - Note AL is the default and does not need to be specified

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	

Conditional execution examples

C source code

```
if (r0 == 0)
{
  r1 = r1 + 1;
}
else
{
  r2 = r2 + 1;
}
```

ARM instructions

unconditional

```
  CMP r0, #0
  BNE else
  ADD r1, r1, #1
  B end
else
  ADD r2, r2, #1
end
  ...
```

- 5 instructions
- 5 words
- 5 or 6 cycles

conditional

```
CMP r0, #0
ADDEQ r1, r1, #1
ADDNE r2, r2, #1
...
```

- 3 instructions
- 3 words
- 3 cycles

Simple Addressing Modes

- Second operand - *<op2>*

  ADD Rd, Rn, *<op2>*

  – Constant
    - `ADD Rd, Rn, #constant  ; Rd = Rn+constant`

  – Shift
    - `ADD R0, R1, LSL #4  ; R0 = R0+(R1*16)`
    - `ADD R0, R1, R2, ASR #4  ; R0 = R1+(R2/16)`

- Memory accessed only with LDR STR
  – Constant in ROM: **=Constant / [PC,#*offs*]**
  – Variable on the stack: **[SP,#*offs*]**
  – Global variable in RAM: **[R*x*]**
  – I/O port: **[R*x*]**

Examples of Conditional Execution

- Use a sequence of several conditional instructions

```
if (a==0) func(1);
    CMP     r0,#0
    MOVEQ   r0,#1
    BLEQ    func
```

- Set the flags, then use various condition codes

```
if (a==0) x=0;
if (a>0)  x=1;
    CMP     r0,#0
    MOVEQ   r1,#0
    MOVGT   r1,#1
```

- Use conditional compare instructions

```
if (a==4 || a==10) x=0;
    CMP     r0,#4
    CMPNE   r0,#10
    MOVEQ   r1,#0
```

ARM Instruction Set

Data Processing Instructions

Data processing Instructions

- Consist of :
  - Arithmetic: `ADD ADC SUB SBC RSB RSC`
  - Logical: `AND ORR EOR BIC`
  - Comparisons: `CMP CMN TST TEQ`
  - Data movement: `MOV MVN`

- These instructions only work on registers, NOT memory.

- Syntax:

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

  - Comparisons set flags only - they do not specify Rd
  - Data movement does not specify Rn

- Second operand is sent to the ALU via barrel shifter.

The Barrel Shifter

LSL : Logical Left Shift

Multiplication by a power of 2

ASR: Arithmetic Right Shift

Division by a power of 2,
preserving the sign bit

LSR : Logical Shift Right

Division by a power of 2

ROR: Rotate Right

Bit rotate with wrap around
from LSB to MSB

RRX: Rotate Right Extended

Single bit rotate with wrap around
from CF to MSB

Branch Instructions

- Branch : `B{<cond>} label`
- Branch with Link : `BL{<cond>} subroutine_label`

- The processor core shifts the offset field left by 2 positions, sign-extends it and adds it to the PC
  - ± 32 Mbyte range
  - How to perform longer branches?

ARM Instruction Set

Load Store Instructions

Register memory architecture

- Allows operations to be performed on (or from) memory, as well as registers.
- If the architecture allows all operands to be in memory or in registers, or in combinations, it is called a "register plus memory" architecture

Load Store Architecture

- **load/store** architecture divides instructions into 2 categories:
  - Memory access (load and store between memory and registers).
  - ALU operations (which only occur between registers).

Load-Store Instructions

Transfer data b/w memory and processor registers.

1. Single Register transfer instructions:

   Data types supported are signed and unsigned words (32 bits), Half-words and bytes.

2. Multiple register transfer instructions:

   Transfer Multiple regiseters b/w memory and the processor in a single instruction.

3. Swap:

   Swaps content of a memory location with the contents of a register.

Load-Store Instructions

LDR	STR	Word
LDRB	STRB	Byte
LDRH	STRH	Halfword
LDRSB		Signed byte load
LDRSH		Signed halfword load

- Memory system must support all access sizes

- Syntax:
  - LDR{<cond>}{<size>} Rd, <address>
  - STR{<cond>}{<size>} Rd, <address>

  e.g. LDREQB

Load-Store Instructions

- Address accessed by LDR/STR is specified by a base register plus an offset
- For word and unsigned byte accesses, offset can be
  - An unsigned 12-bit immediate value (ie 0 - 4095 bytes).

```
LDR r0,[r1,#8]
```

  - A register, optionally shifted by an immediate value

```
LDR r0,[r1,r2]
LDR r0,[r1,r2,LSL#2]
```

- This can be either added or subtracted from the base register:

```
LDR r0,[r1,#-8]
LDR r0,[r1,-r2]
LDR r0,[r1,-r2,LSL#2]
```

- For halfword and signed halfword / byte, offset can be:
  - An unsigned 8 bit immediate value (ie 0-255 bytes).
  - A register (unshifted).
- Choice of *pre-indexed* or *post-indexed* addressing

LDM/STM Operations

- Syntax:

  `<LDM|STM>`{<cond>}<addressing_mode> Rb{!}, <register list>

- 4 addressing modes:

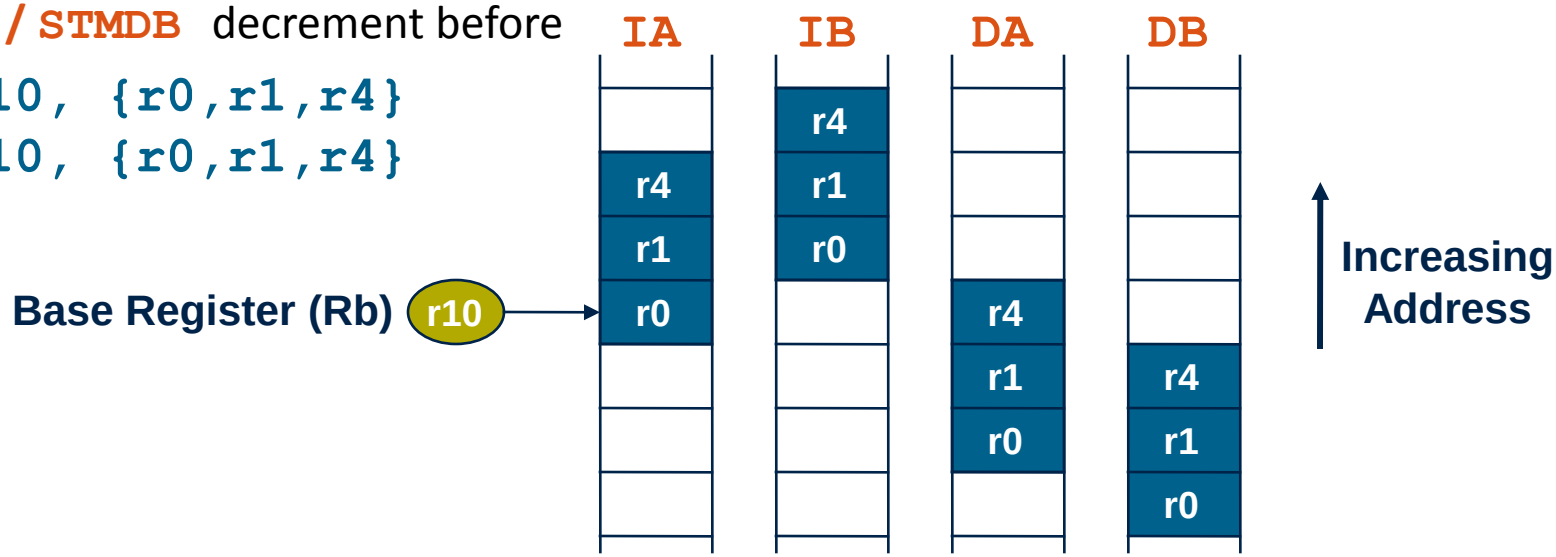
  `LDMIA / STMIA` increment after

  `LDMIB / STMIB` increment before

  `LDMDA / STMDA` decrement after

  `LDMDB / STMDB` decrement before

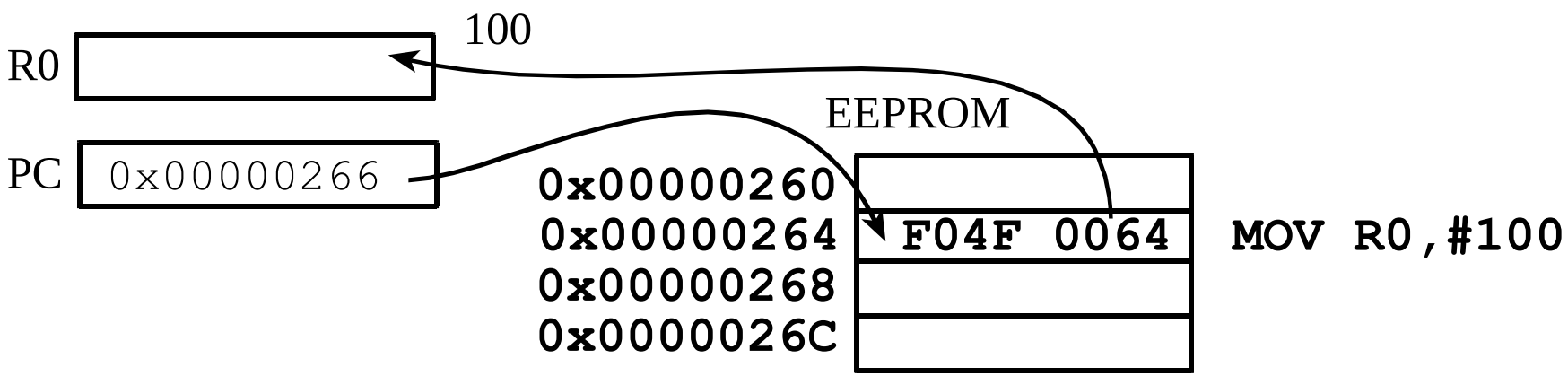
```
LDMxx r10, {r0,r1,r4}
STMxx r10, {r0,r1,r4}
```



Addressing Modes

- Immediate addressing
  - Data is contained in the instruction

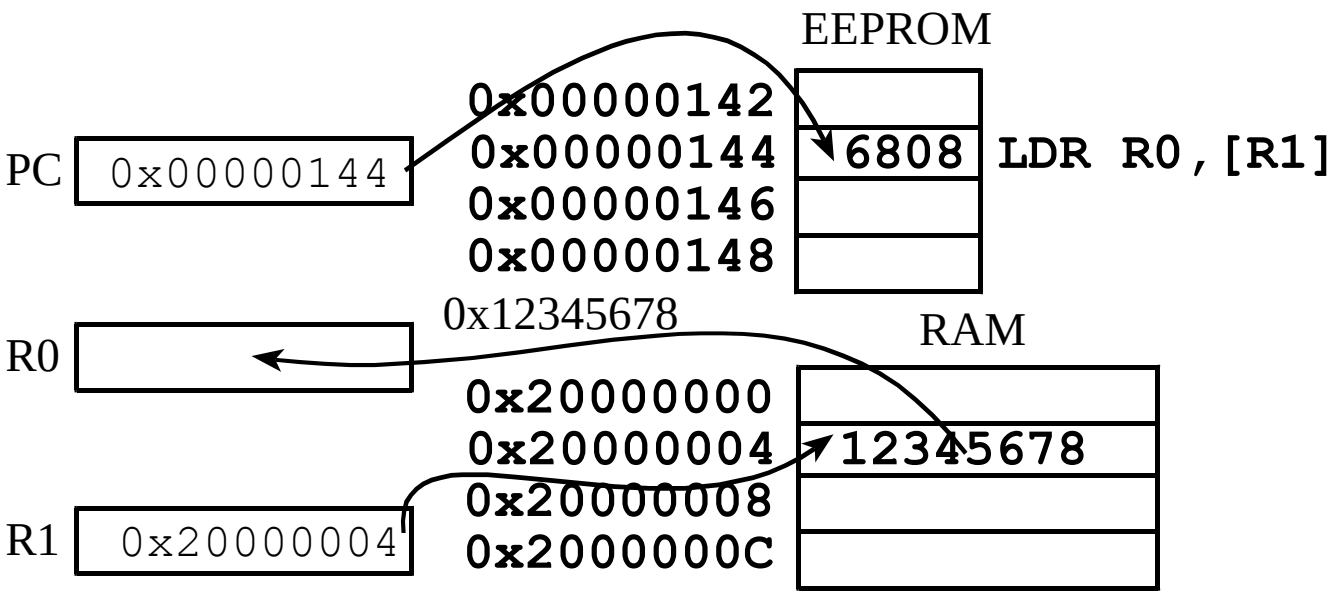
```
MOV    R0,#100        ; R0=100, immediate addressing
```



Addressing Modes

- Indexed Addressing
  - Address of the data in memory is in a register

```
LDR    R0,[R1]        ; R0= value pointed to by R1
```



Memory Access Instructions

- Loading a register with a constant, address, or data
  - `LDR  Rd, =number`
  - `LDR  Rd, =label`

- **`LDR`** and **`STR`** used to load/store RAM using register-indexed addressing
  - **`Register [R0]`**
  - **`Base address plus offset [R0,#16]`**

Load/Store Instructions

- General load/store instruction format

```
LDR{type} Rd,[Rn]       ;load memory at [Rn] to Rd
STR{type} Rt,[Rn]       ;store Rt to memory at [Rn]
LDR{type} Rd,[Rn, #n]  ;load memory at [Rn+n] to Rd
STR{type} Rt,[Rn, #n]  ;store Rt to memory [Rn+n]
LDR{type} Rd,[Rn,Rm,LSL #n]  ;load [Rn+Rm<<n] to Rd
STR{type} Rt,[Rn,Rm,LSL #n]  ;store Rt to [Rn+Rm<<n]
```

{type}	*Data type*	*Meaning*	
	32-bit word	0 to 4,294,967,295	or -2,147,483,648 to +2,147,483,647
B	Unsigned 8-bit byte	0 to 255,	Zero pad to 32 bits on load
SB	Signed 8-bit byte	-128 to +127,	Sign extend to 32 bits on load
H	Unsigned 16-bit halfword	0 to 65535,	Zero pad to 32 bits on load
SH	Signed 16-bit halfword	-32768 to +32767,	Sign extend to 32 bits on load
D	64-bit data		Uses two registers

Implementation of local variable using a stack frame

```
void sub(void)
{
    short y1,y2,y3;   /* 3 local variables*/
    y1=1000;
    y2=2000;
    y3=y1+y2;
}
```

http://192.168.1.3/stack1.gif

How parameters are passed in C

```
int x1;
static int x2;
const int x3=1000;

int add3(int z1, int z2, int z3){ int y;
    y=z1+z2+z3;
    return(y);}

void main(void){ int y;
    x1=1000;
    x2=1000;
    y=add3(x1,x2,x3);
```

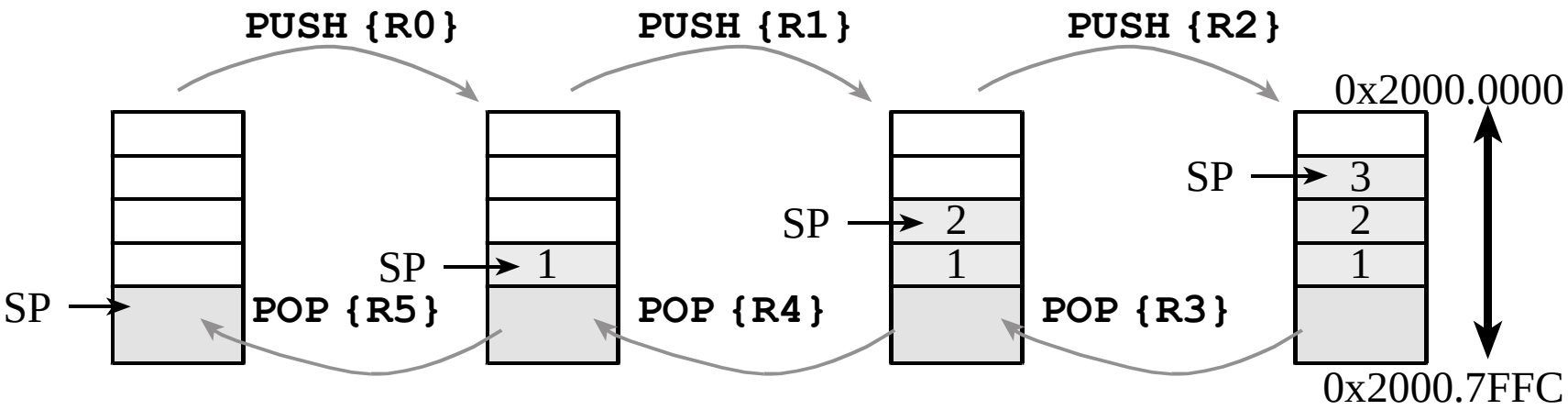
http://192.168.1.3/stack.gif

The Stack

- Stack is last-in-first-out (LIFO) storage
  - 32-bit data
- Stack pointer, SP or R13, points to top element of stack
- Stack pointer *decremented* as data placed on stack
- **`PUSH`** and **`POP`** instructions used to load and retrieve data

The Stack

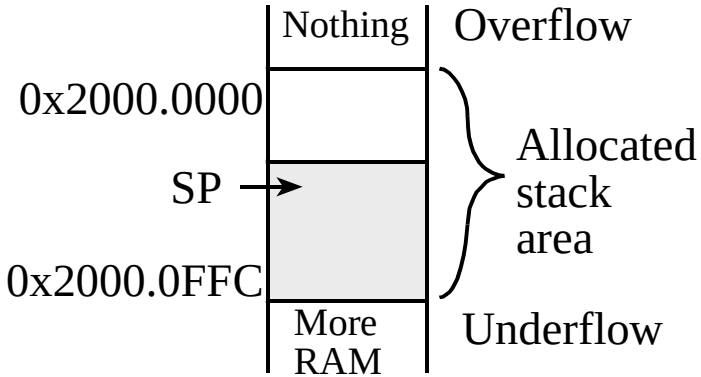
- Stack is last-in-first-out (LIFO) storage
  - 32-bit data
- Stack pointer, SP or R13, points to top element of stack
- Stack pointer *decremented* as data placed on stack (*incremented* when data is removed)
- **`PUSH`** and **`POP`** instructions used to load and retrieve data



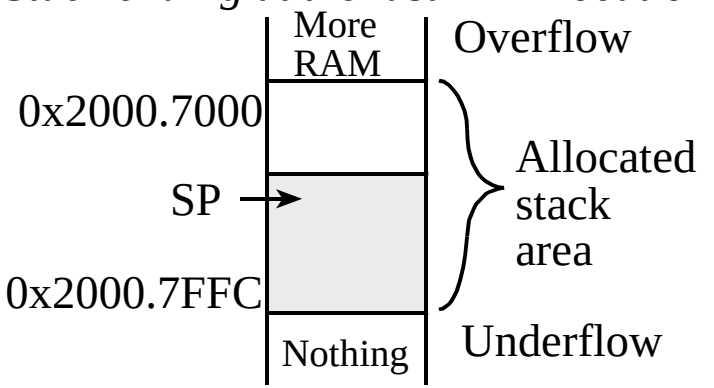
The Stack Usage

- Stack memory allocation

Stack starting at the first RAM location

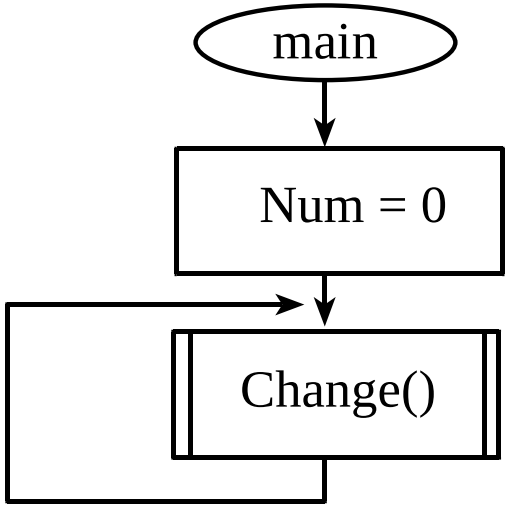


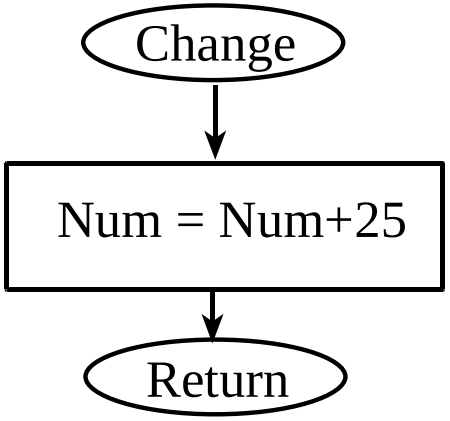
Stack ending at the last RAM location



- Rules for stack use
  - Stack should always be balanced, i.e. functions should have an equal number of pushes and pops
  - Stack accesses (push or pop) should not be performed outside the allocated area
  - Stack reads and writes should not be performed within the free area

Functions






```
Change LDR   R1,=Num    ; 5) R1 = &Num
       LDR   R0,[R1]    ; 6) R0 = Num
       ADD   R0,R0,#25  ; 7) R0 = Num+25
       STR   R0,[R1]    ; 8) Num = Num+25
       BX    LR         ; 9) return
main   LDR   R1,=Num    ; 1) R1 = &Num
       MOV   R0,#0      ; 2) R0 = 0
       STR   R0,[R1]    ; 3) Num = 0
loop   BL    Change     ; 4) function call
       B     loop       ; 10) repeat
```

```
unsigned long Num;
void Change(void){
  Num = Num+25;
}
void main(void){
  Num = 0;
  while(1){
    Change();
  }
}
```

Subroutines

```
;------------Rand100------------
; Return R0=a random number between
; 1 and 100. Call Random and then divide
; the generated number by 100
; return the remainder+1
Rand100
       PUSH {LR}  ; SAVE Link
       BL   Random
    ;R0 is a 32-bit random number
       LDR  R1,=100
       BL   Divide
       ADD  R0,R3,#1
       POP  {LR}  ;Restore Link back
       BX   LR
```

POP {PC}

```
;------------Divide------------
; find the unsigned quotient and remainder
; Inputs:  dividend in R0
;          divisor in R1
; Outputs: quotient in R2
;          remainder in R3
;dividend = divisor*quotient + remainder
Divide
  UDIV R2,R0,R1    ;R2=R0/R1,R2 is quotient
  MUL  R3,R2,R1    ;R3=(R0/R1)*R1
  SUB  R3,R0,R3    ;R3=R0%R1,
                   ;R3 is remainder of R0/R1
  BX   LR          ;return

       ALIGN
       END
```

One function calls another, so LR must be saved

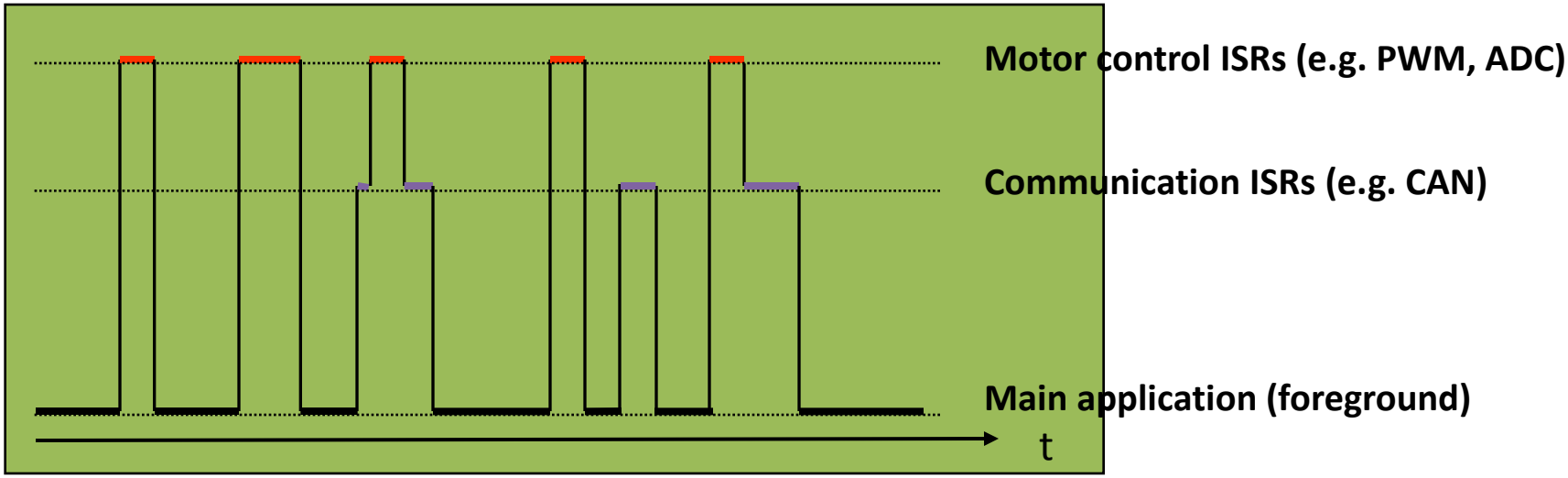
Reset, Subroutines and Stack

- A ***Reset*** occurs immediately after power is applied and when the reset signal is asserted (Reset button pressed)
- The Stack Pointer, SP (R13) is initialized at ***Reset*** to the 32-bit value at location 0 (Default: 0x20000408)
- The Program Counter, PC (R15) is initialized at ***Reset*** to the 32-bit value at location 4 (Reset Vector)
- The Link Register (R14) is initialized at Reset to 0xFFFFFFFF
- Thumb bit is set at *Reset*
- Processor automatically saves return address in LR when a subroutine call is invoked.
- User can push and pull multiple registers on or from the ***Stack*** at subroutine entry and before subroutine return.

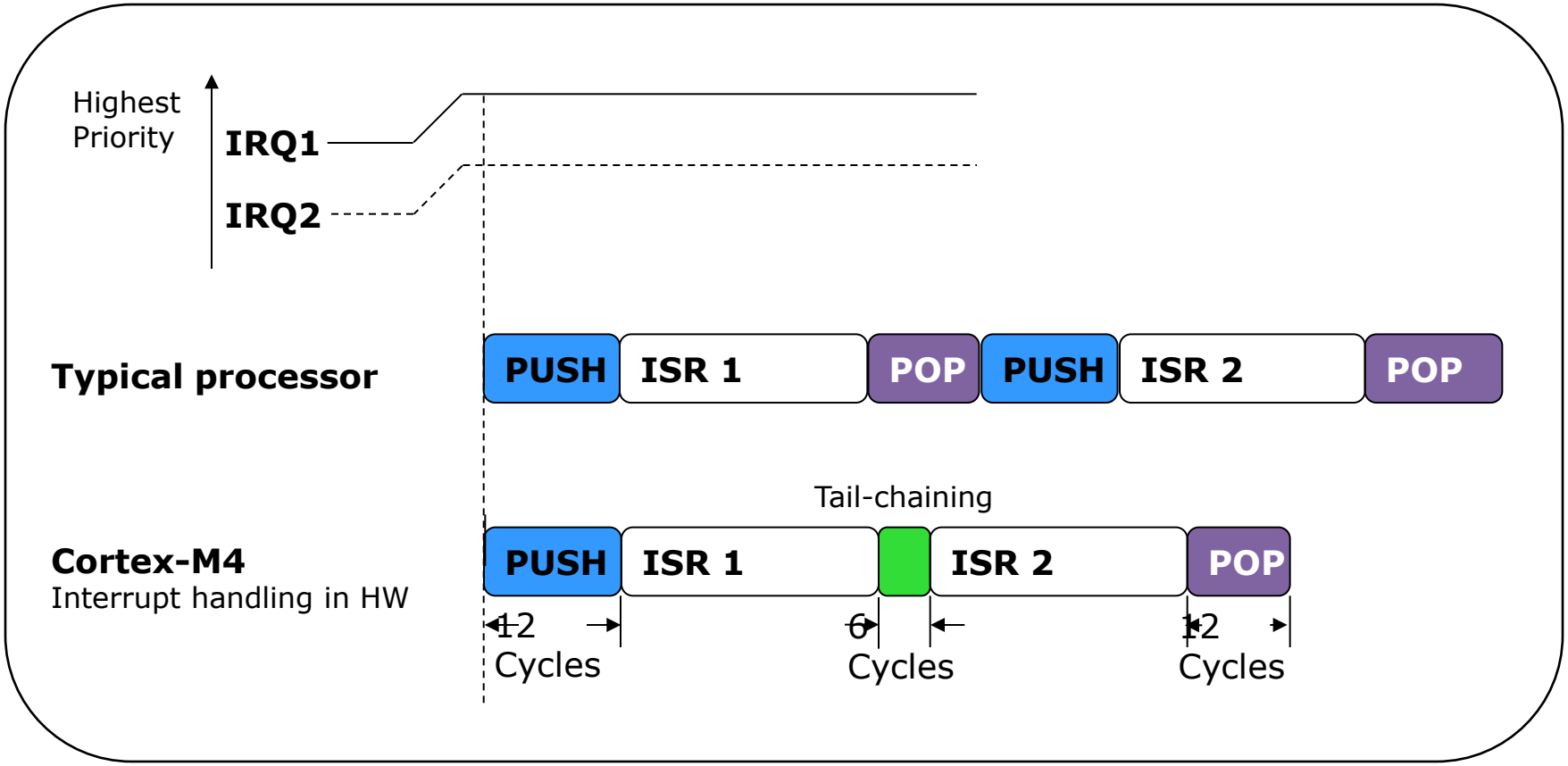
NVIC

Nested Vectored Interrupt Controller (NVIC)

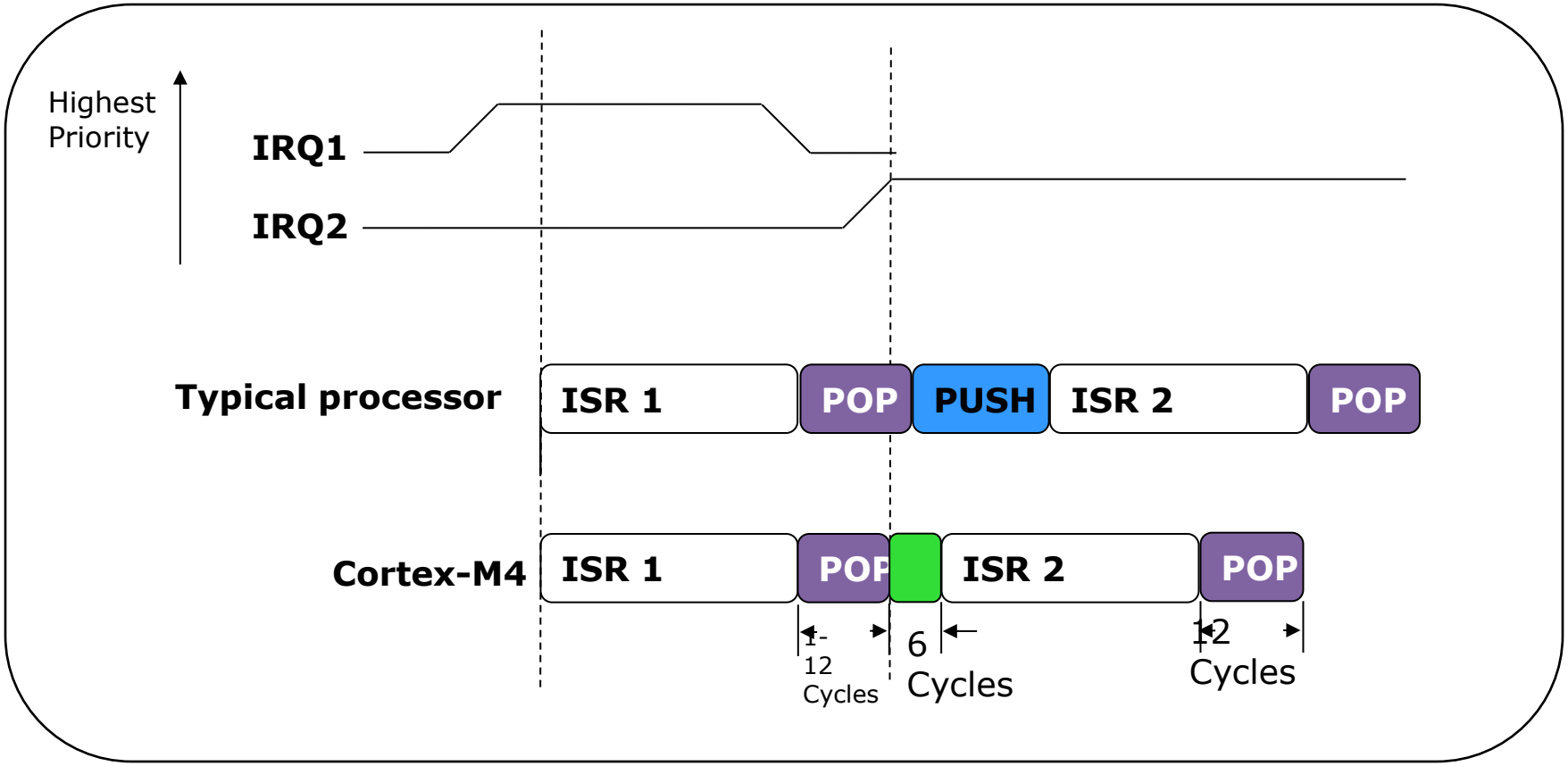
- Handles exceptions and interrupts
- 8 programmable priority levels, priority grouping
- 7 exceptions and 71 Interrupts
- Automatic state saving and restoring
- Automatic reading of the vector table entry
- Pre-emptive/Nested Interrupts
- Tail-chaining
- Deterministic: always 12 cycles or 6 with tail-chaining



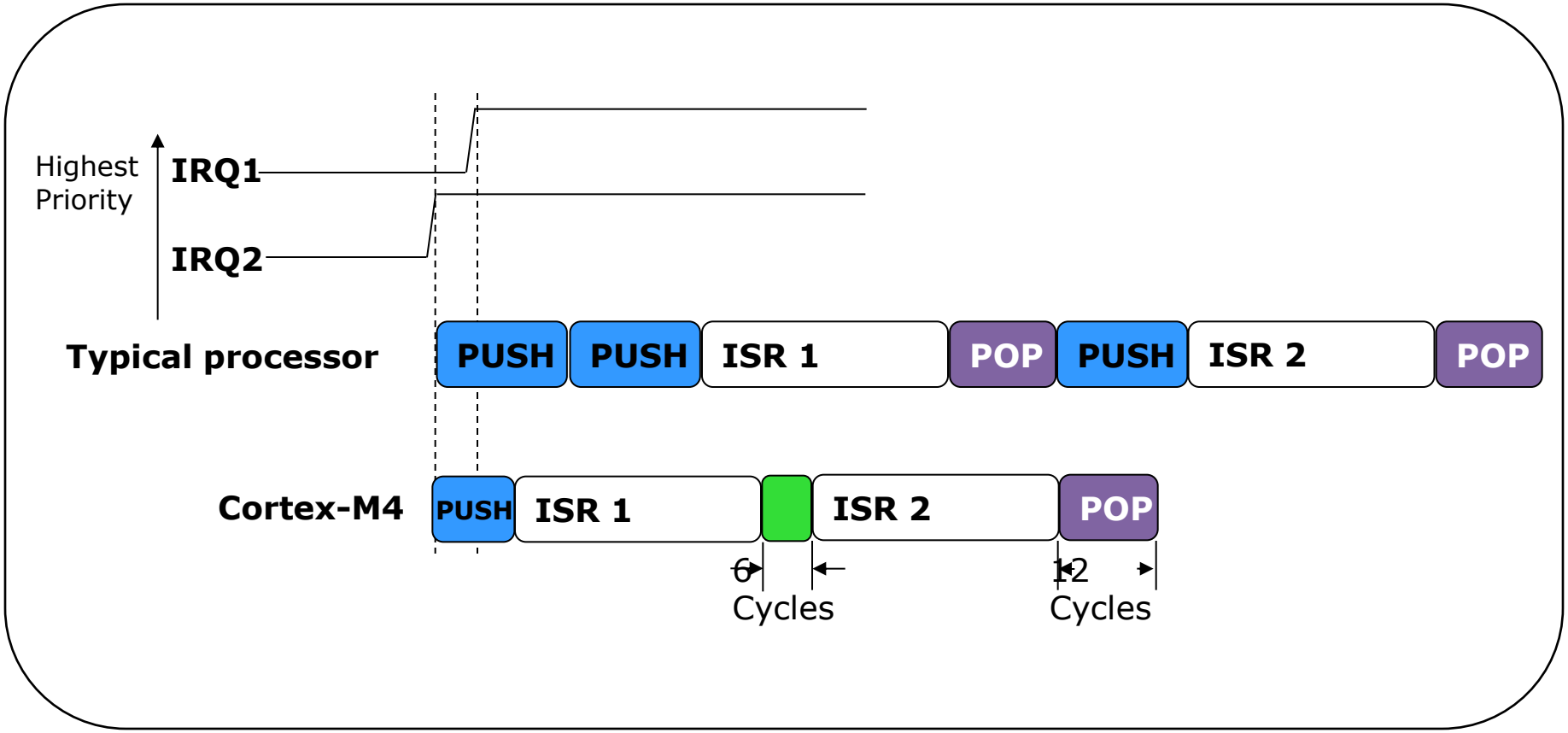
Interrupt Latency - Tail Chaining



Interrupt Latency – Pre-emption



Interrupt Latency – Late Arrival



Cortex-M4® Interrupt Handling

Interrupt handling is automatic. No instruction overhead.

Entry

- Automatically pushes registers R0–R3, R12, LR, PSR, and PC onto the stack
- In parallel, ISR is pre-fetched on the instruction bus. ISR ready to start executing as soon as stack PUSH complete

Exit

- Processor state is automatically restored from the stack
- In parallel, interrupted instruction is pre-fetched ready for execution upon completion of stack POP

Vector Table for ARMv7-M

- First entry contains initial Main SP
- All other entries are addresses for exception handlers
  - Must always have LSBit = 1 (for Thumb)
- Table has up to 496 external interrupts
  - Implementation-defined
  - Maximum table size is 2048 bytes
- Table may be relocated
  - Use Vector Table Offset Register
  - Still require minimal table entries at 0x0 for booting the core
- Each exception has a vector number
  - Used in Interrupt Control and State Register to indicate the active or pending exception type
- Table can be generated using C code
  - Example provided later

Address		Vector #
0x40 + 4*N	External N	16 + N
…	…	…
0x40	External 0	16
0x3C	SysTick	15
0x38	PendSV	14
0x34	Reserved	13
0x30	Debug Monitor	12
0x2C	SVC	11
0x1C to 0x28	Reserved (x4)	7-10
0x18	Usage Fault	6
0x14	Bus Fault	5
0x10	Mem Manage Fault	4
0x0C	Hard Fault	3
0x08	NMI	2
0x04	Reset	1
0x00	Initial Main SP	N/A

Cortex-M4® Vector Table

- After reset, vector table is located at address 0
- Each entry contains the address of the function to be executed
- The value in address 0x00 is used as starting address of the Main Stack Pointer (MSP)
- Vector table can be relocated by writing to the VTABLE register
  (must be aligned on a 1KB boundary)
- Open startup_ccs.c to see vector table coding

Exception number	IRQ number	Offset	Vector
154	138	0x0268	IRQ131
. . .		0x004C	. . .
18	2	0x0048	IRQ2
17	1	0x0044	IRQ1
16	0	0x0040	IRQ0
15	-1	0x003C	Systick
14	-2	0x0038	PendSV
13			Reserved
12			Reserved for Debug
11	-5	0x002C	SVCall
10			Reserved
9			
8			
7			
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
		0x0000	Initial SP value

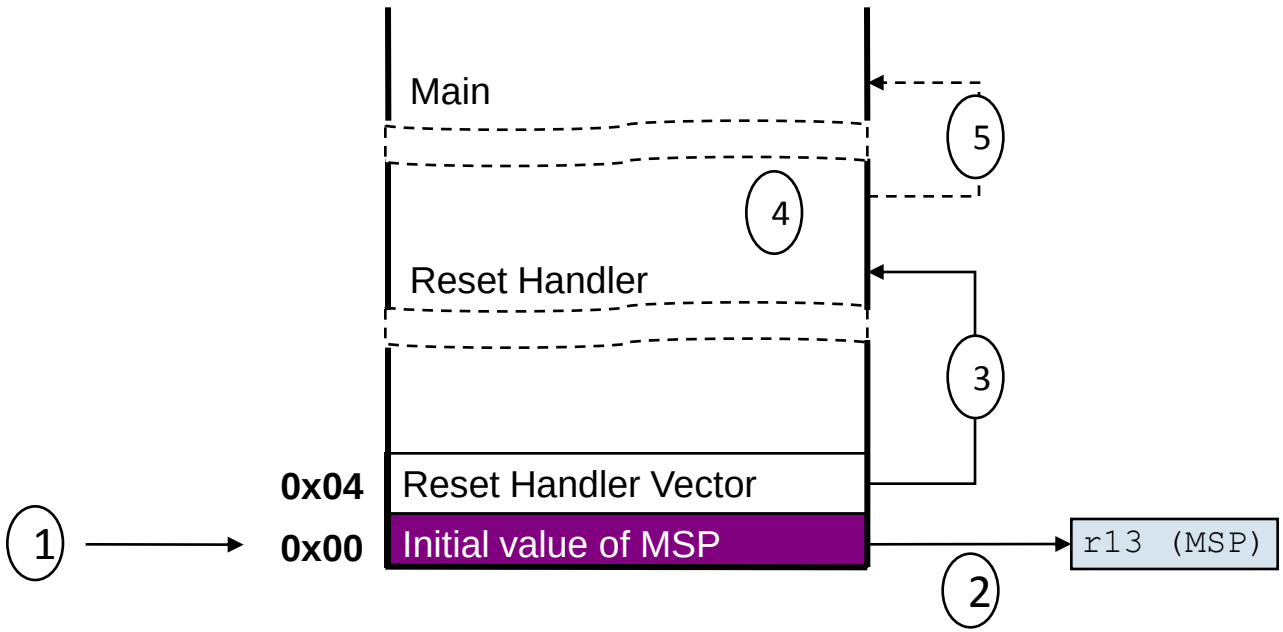
Cortex-M4® Exception Types

Vector Number	Exception Type	Priority	Vector address	Descriptions
1	Reset	-3	0x04	Reset
2	NMI	-2	0x08	Non-Maskable Interrupt
3	Hard Fault	-1	0x0C	Error during exception processing
4	Memory Management Fault	Programmable	0x10	MPU violation
5	Bus Fault	Programmable	0x14	Bus error (Prefetch or data abort)
6	Usage Fault	Programmable	0x18	Exceptions due to program errors
7-10	Reserved	-	0x1C - 0x28	
11	SVCall	Programmable	0x2C	SVC instruction
12	Debug Monitor	Programmable	0x30	Exception for debug
13	Reserved	-	0x34	
14	PendSV	Programmable	0x38	
15	SysTick	Programmable	0x3C	System Tick Timer
16 and above	Interrupts	Programmable	0x40	External interrupts (Peripherals)

Vector Table in Assembly

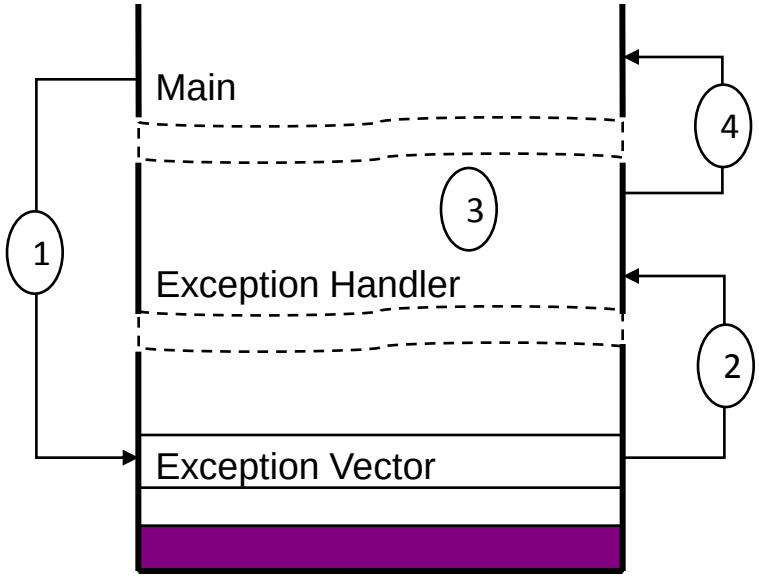
```
                PRESERVE8
                THUMB
                IMPORT ||Image$$ARM_LIB_STACK$$ZI$$Limit||
                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors

__Vectors       DCD     ||Image$$ARM_LIB_STACK$$ZI$$Limit||  ; Top of Stack
                DCD     Reset_Handler                ; Reset Handler
                DCD     NMI_Handler                  ; NMI Handler
                DCD     HardFault_Handler            ; Hard Fault Handler
                DCD     MemManage_Handler            ; MemManage Fault Handler
                DCD     BusFault_Handler             ; Bus Fault Handler
                DCD     UsageFault_Handler           ; Usage Fault Handler
                DCD     0, 0, 0, 0,                  ; Reserved x4
                DCD     SVC_Handler,                 ; SVCall Handler
                DCD     Debug_Monitor                ; Debug Monitor Handler
                DCD     0                            ; Reserved
                DCD     PendSV_Handler               ; PendSV Handler
                DCD     SysTick_Handler              ; SysTick Handler
                ; External vectors start here
```

Reset Behaviour



1. A reset occurs (Reset input was asserted)
2. Load MSP (Main Stack Pointer) register initial value from address 0x00
3. Load reset handler vector address from address 0x04
4. Reset handler executes in Thread Mode
5. Optional: Reset handler branches to the main program

Exception Behaviour



1. Exception occurs
   - Current instruction stream stops
   - Processor accesses vector table
2. Vector address for the exception loaded from the vector table
3. Exception handler executes in Handler Mode
4. Exception handler returns to main