EMBEDDED SOFTWARE DEVELOPMENT

- Several aspects define the target machine for an Embedded Application
  - Hardware
    - ISA
    - Memory organization
    - Peripherals (IRQ controllers, Timers, I/O controllers, GPIO, ADCs, DACs, …)
  - A System on a Chip (SoC) provides a convenient solution for most embedded applications on the hardware front

EMBEDDED SOFTWARE DEVELOPMENT

- Several aspects define the target machine for an Embedded Application Software
  - Software
    - Programming language(s)
      - medium- and high-level languages are mostly adopted (C/C++) but the use of assembly is sometimes unavoidable
    - Operating System (if any)
    - Embedded Application Binary Interface (EABI)
      - Similar to ABI for general purpose computing
      - Describes
        - Calling conventions: how to pass parameters to subroutines and get back return values (eg. AAPCS)
        - System calls
        - The binary format of object files and libraries (e.g. ELF)
        - Native data types: size, organization, alignments, endianness (big-endian, little-endian)

EMBEDDED SOFTWARE DEVELOPMENT

- Several programming models are available to develop embedded programs for a given target machine
  - Compilation on the target machine
  - Cross-compilation
  - Interactive environment on the target machine

EMBEDDED SOFTWARE DEVELOPMENT

- Compilation on the target machine
  - A software toolchain (e.g. GCC) is used on the target machine to obtain executable code
  - The executable runs on the same machine it was compiled on
  - Requires an OS to support the toolchain
  - **Not suitable for resource-constrained machines**

EMBEDDED SOFTWARE DEVELOPMENT

- Cross-compilation
  - A software toolchain (e.g. GCC) is used on a development machine to produce code for the target machine
  - The executable is loaded on the target machine and runs there. OSs are often used but are not mandatory (bare metal)
  - Emulators can be used on the development machine to run the target code
  - **Most used model for resource-constrained machines**

EMBEDDED SOFTWARE DEVELOPMENT

- Interactive environment on the target machine
  - An interpreter is executed on the target machine: code can be easily tested on the target, experimental programming is feasible
  - Depending on the complexity of the interpreter (e.g. Python), **this model could be not suitable for resource-constrained machines**
  - **Forth interpreters can be implemented easily for resource-constrained machines using no OS (bare metal)**

EMBEDDED SOFTWARE DEVELOPMENT

- Calling conventions
    - To write software for ARM (32-bit) we follow the Procedure Call Standard for the ARM® Architecture (AAPCS)

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Table 2, Core registers and AAPCS usage

Procedure Call Standard for the ARM® Architecture (AAPCS) p. 14 - (IHI0042F_aapcs.pdf)

EMBEDDED SOFTWARE DEVELOPMENT

- Calling conventions
  - To write software for ARM (32-bit) we follow the Procedure Call Standard for the ARM® Architecture (AAPCS)
  - The basics:
    - The first four registers r0-r3 (a1-a4) are used to pass argument values into a subroutine and to return a result value from a function. They may also be used to hold intermediate values within a routine (but, in general, only between subroutine calls)
    - Registers r4-r8, r10, and r11 (v1-v5, v7 and v8) are used to hold the values of a routine's local variables
    - A subroutine must preserve the contents of registers r4-r8, r10, r11 and SP
    - A double-word sized type is passed in two consecutive registers (e.g., r0 and r1, or r2 and r3)
    - The stack is a contiguous area of memory that may be used for storage of local variables and for passing additional arguments to subroutines when there are insufficient argument registers available. Eight byte stack alignment is a requirement of the AAPCS

Procedure Call Standard for the ARM® Architecture (AAPCS) p. 15 - (IHI0042F_aapcs.pdf)

EMBEDDED SOFTWARE DEVELOPMENT

- Result return and parameter passing for 32-bit and 64-bit values
  - Result Return
    - A Fundamental Data Type that is smaller than 4 bytes is zero- or sign-extended to a word and returned in r0
    - A word-sized Fundamental Data Type (e.g., int, float) is returned in r0
    - A double-word sized Fundamental Data Type (e.g.,long long, double and 64-bit containerized vectors) is returned in r0 and r1
    - A Composite Type not larger than 4 bytes is returned in r0. The format is as if the result had been stored in memory at a word-aligned address and then loaded into r0 with an LDR instruction. Any bits in r0 that lie outside the bounds of the result have unspecified values
    - A Composite Type larger than 4 bytes, or whose size cannot be determined statically by both caller and callee, is stored in memory at an address passed as an extra argument when the function was called.The memory to be used for the result may be modified at any point during the function call
  - Parameter Passing
    - The base standard provides for passing arguments in core registers (r0-r3) and on the stack. For subroutines that take a small number of parameters, only registers are used, greatly reducing the overhead of a call

Procedure Call Standard for the ARM® Architecture (AAPCS) p. 18-19 - (IHI0042F_aapcs.pdf)

EMBEDDED SOFTWARE DEVELOPMENT

- Calling conventions, like AAPCS, make it possible linking object code from multiple sources (e.g. object code produced by different compilers, even for different languages, libraries) and exchanging data with the OS
  - first.s (ARM Assembly, Raspberry Pi, GNU/Linux)

```
pi@raspberrypi: 1st$ cat first.s
/* -- first.s */
/* This is a comment */
.global main /* 'main' is our entry point and must be global */

main:          /* This is main */
    mov r0, #2 /* Load 2 into register r0 */
    bx lr      /* Return from main */
pi@raspberrypi: 1st$ make
as -o first.o first.s
gcc -o first first.o
pi@raspberrypi: 1st$ ./first
pi@raspberrypi: 1st$ echo $?
2
pi@raspberrypi:
```

VALUE IN R0 IS THE RETURN VALUE FOR THE MAIN FUNCTION AND SO FOR THE PROGRAM

EXAMPLE 1 (ARM ASM, RASPBERRY PI, GNU/LINUX)

- Write an assembly program that loads 2 in register r1, 3 in r2 and puts r1+r2 into r3 then enters a never ending loop
  - This deals with basic ARM assembly syntax, global values, program entry point, and assembly compiling through the C toolchain

EXAMPLE 1 (ARM ASM, RASPBERRY PI, GNU/LINUX)

```
@ Example 1
@
@ compile with:
@ cc -g 1.s -o 1

.global main                @ symbol main must be exported for linking with
                            @ the C-runtime (crt)

main:
        mov     r1, #2              @ assign 2 to r1
        mov     r2, #3              @ assign 3 to r2
        add     r3, r1, r2          @ assign r1+r2 to r3
loop:
        b       loop                @ never ending loop
```

EXAMPLE 2 (ARM ASM, RASPBERRY PI, GNU/LINUX)

- Write an assembly program that loads 3 in register r1, puts r1*5 into r3 then terminates using the value contained in r3 as exit code
    - This deals with system calls: the hardware support provided in the ISA and the conventions of the Operating System. It also involves the use of the barrel shifter in data transfer instructions.

EXAMPLE 2 (ARM ASM, RASPBERRY PI, GNU/LINUX)

```
@ Example 2
@
@ assemble with:
@ as -g 2.s -o 2.o
@ ld 2.o -o 2

.global _start  @ export

_start:
        mov     r1, #3
        add     r3, r1, r1, LSL #2  @ r3=r1+r1*4=r1
        mov     r0, r3          @ the argument for the exit system call must be in r0
        mov     r7, #1          @ r7=1, this selects the exit system call
        svc     0               @ SuperVisor Call
```

EXAMPLE 3 (ARM ASM, RASPBERRY PI, GNU/LINUX)

- Write an assembly program that loads **`4096`** into register **`r0`**, **`6000`** in **`r1`**, puts **`r0*r1`** (32-bit multiplication) into **`r0`**, then terminates using the value contained in **`r0`** as exit code
    - This is about loading large constant values into registers, constants definitions, assembler changing **`ldr`** into **`mov`**, 32-bit multiplication, and exit system call reducing ints to 8-bit values. Use GDB to verify code
- We need real I/O!

EXAMPLE 3 (ARM ASM, RASPBERRY PI, GNU/LINUX)

```
@ Example 3
@
@ assemble with:
@
@ as -g 3.s -o 3.o
@ ld 3.o -o 3
@

@
@ Definitions
@
@ void exit(int status)
SYSCALL_EXIT=1

.global _start

_start:
        ldr r0, =#4096
        ldr r1, =#6000
        mul r0, r1, r0
        mov r7, #SYSCALL_EXIT
        svc 0    @ SuperVisor Call
```

EXAMPLE 4 (ARM ASM, RASPBERRY PI, GNU/LINUX)

- Write an assembly program that prints a message (e.g. "**`Hello world!`**"), checks that all the bytes have been put out, and exits with the proper code
    - This is about using the write system call with **`stdout`**, referencing local values with synonyms, section markers, definition of constant strings in data section, and expressions

EXAMPLE 4 (ARM ASM, RASPBERRY PI, GNU/LINUX)

```
@ Example 4
@
@ assemble with:
@
@ as -g 4.s -o 4.o
@ ld 4.o -o 4
@

@
@ Definitions:
@
@ void exit(int status)
@
SYSCALL_EXIT=1
@
@ ssize_t write(int fd, const void *buffer, size_t size)
@
SYSCALL_WRITE=4
@
@ each process finds the three stdio file descriptors
@ open at start:
@
@ 0: stdin
```

EXAMPLE 4 (ARM ASM, RASPBERRY PI, GNU/LINUX)

```
@ 1: stdout
@ 2: stderr
@
stdout=1

.text

.global _start
_start:
        mov     r0, #stdout
        ldr     r1, =message
        ldr     r2, =message_len
        mov v1, r2
        mov r7, #SYSCALL_WRITE
        svc 0

        cmp r0, v1
        moveq r0, $0

@ exit(int status):
        mov     r7, #SYSCALL_EXIT
        svc     0
```

EXAMPLE 4 (ARM ASM, RASPBERRY PI, GNU/LINUX)

```
.data

message:
        .ascii "Hello World!\n"
message_len= .-message
@
@ . evaluates to the current address: the address where the next instruction
@ or data is to be assembled. In this case the address is that of the byte
@ following the last character of the message string.
@ If x is the address where the first character of the string is
@ stored in memory:
@ message=x
@ message_len=.-message=(x+13)-message=(x+13)-x=13
```

EXAMPLE 5 (ARM ASM, RASPBERRY PI, GNU/LINUX)

- Write an assembly program that loads **`4096`** into register **`r0`**, **`6000`** in **`r1`**, puts **`r0*r1`** (32-bit multiplication) into **`r2`**, prints the value in **`r2`** using **`printf()`**, then terminates using the value returned by **`printf()`** as exit code
  - This is about storing and restoring multiple registers and keeping stack 8-byte aligned when calling functions, referencing call arguments with synonyms, using the C standard library `printf()` function from assembly, and passing a return value from main

EXAMPLE 5 (ARM ASM, RASPBERRY PI, GNU/LINUX)

```
@ Example 5
@
@ compile with:
@
@ cc -g 5.s -o 5
@
@

.text
.global main

main:
        push {ip, lr}
        @ push {ip, lr} is a pseudo-instruction that stores
        @ the values of ip and lr on the stack in a precise
        @ order that simply follows that of register numbers.
        @ This way there is no need to keep track of pushes
        @ as with other ISAs.
        @ The value of ip is not important, but it is stored
        @ too because stack must be kept 8-byte aligned when
        @ calling other functions

        ldr     r0, =#4096
```

EXAMPLE 5 (ARM ASM, RASPBERRY PI, GNU/LINUX)

```
        ldr     r1, =#6000
        mul     a2, r0, r1

        ldr     a1, =format
        bl      printf

        pop {ip, lr} @ restoring registers is as simple as storing them
        bx      lr
.data
format:
        .asciz "value: %u\n"
```

EXAMPLE 6 (ARM ASM, RASPBERRY PI, GNU/LINUX)

- Write an assembly program that prints the integer values from 10 to 0
  - This is about loops in assembly and using synonyms to reference local values

EXAMPLE 6 (ARM ASM, RASPBERRY PI, GNU/LINUX)

```
@ Example 6
@
@ compile with:
@
@ cc -g 6.s -o 6
@
@

.text
.global main

main:
        push {ip, lr}
@
@       Equivalent C code:
@
@       int i;
@
@       i=10;
@       do{
@               printf("%i\n", i);
@               --i;
@       }while(i>=0);
```

EXAMPLE 6 (ARM ASM, RASPBERRY PI, GNU/LINUX)

```
@

        ldr v1, =#10            @ i=10 -> v1=10

loop:
        ldr a1, =format         @
        mov a2, v1              @       printf(format, v1)
        bl  printf              @

        subs v1, v1, #1         @       --i; -> v1=v1-1 CPSR.N=(v1<0)
        bpl loop                @       }while(!CPSR.N);

        pop {ip, lr}
        bx      lr

.data
format:
        .asciz "%i\n"
```

EXAMPLE 7 (ARM ASM, RASPBERRY PI, GNU/LINUX)

- Write an assembly program that computes the sum of **`i*400000000 (i=10,…,0`**) printing each product and the final result
    - This is about loops in assembly and using synonyms to reference local values

EXAMPLE 7 (ARM ASM, RASPBERRY PI, GNU/LINUX)

```
@ Example 7
@
@ compile with:
@
@ cc -g 7.s -o 7
@
@

.text
.global main

main:
        push {ip, lr}

@       Equivalent C code
@
@       long long unsigned a; /* 64-bit value */
@       unsigned c; /* 32-bit value */
@       unsigned i; /* 32-bit value */
@
@       a=0;
@       c=400000000
@       i=10;
```

EXAMPLE 7 (ARM ASM, RASPBERRY PI, GNU/LINUX)

```
@       do{
@               a+=i*c;
@               --i;
@       }while(i+1>0);
@

        ldr     v1, =#10        @ i=10;
        ldr v2, =#400000000     @ c=400000000;
        mov v3, #0              @ a=0; (lowest 32 bits)
        mov v4, #0              @ a=0; (highest 32 bits)
loop:
        umlal v3,v4,v1,v2       @ a+=i*c ->

        ldr a1, =format1        @
        mov a2, v1              @ printf(format1, i, c);
        mov a3, v2              @
        bl  printf              @

        subs v1, v1, #1         @ --i -> v1=v1-1, CPSR.N=(v1<0)
        bpl loop                @ while(!CPSR.N)

        ldr a1, =format2        @
        mov a3, v3              @ printf(format2, a);
```

EXAMPLE 7 (ARM ASM, RASPBERRY PI, GNU/LINUX)

```
        mov a3, v3                @ printf(format2, a);
        mov a4, v4                @ the second arg to printf is 64-bit wide
        bl  printf                @ a couple of contiguous register is needed
                                  @ a1 is used for the first arg so the first
                                  @ (and last) usable couple of registers to
                                  @ hold a 64-bit arg to printf is
                                  @ a3,a4
                                  @ In this case any other argument would be stored
                                  @ on the stack (try adding a %x to format2 and see
                                  @ what happens...

        pop {ip, lr}
        bx      lr

.data
format1:
        .asciz "%u*%u\n"
format2:
        .asciz "=%llu\n"
```

EXAMPLE 8 (ARM ASM, RASPBERRY PI, GNU/LINUX)

- Write a program composed of a module written in C and one in assembly. The former contains the **`main()`** function which allocates two arrays and uses the assembly-defined **`void array_copy(int *src, int *dst, size_t size)`** function to copy values from the first array to the second. Define the function **`void array_print(int *array, size_t size)`** in C to conveniently print the array before and after the copy.
    - This is about calling an assembly-defined function from a C function, load and store instructions, post-indexed addressing

EXAMPLE 8 (ARM ASM, RASPBERRY PI, GNU/LINUX)

```
pi@raspberrypi: 08$ cat 8_main.c
/* Example 8
 *
 * compile with:
 *
 * cc -g 8.s 8_main.c -o 8
 *
 */

#include <stdio.h>
#include <stdlib.h>

/* Protos */

void
array_copy(int *src, int *dst, size_t
size);

void
array_print(int *array, size_t size);

/* Functions */

void
array_print(int *array, size_t size){
```

```
        size_t i;

        for(i=0; i<size; ++i){
                printf("%i ", array[i]);
        }
        printf("\n");
}

int
main(void){
        int a[]={1,2,3};
        int b[3];
        size_t size;

        size=sizeof(a)/sizeof(int);
        printf("Source array:\n");
        array_print(a, size);
        printf("Destination array before
copy:\n");
        array_print(b, size);
        array_copy(a, b, size);
        printf("Destination array after
copy:\n");
        array_print(b, size);
        return 0;
}
```

EXAMPLE 8 (ARM ASM, RASPBERRY PI, GNU/LINUX)

```
pi@raspberrypi: 08$ cat 8.s
@ Example 8
@
@ compile with:
@
@ cc -g 8.s 8_main.c -o 8
@
@

.text
.global array_copy

array_copy:

loop:   ldr v1, [a1], #4
        str v1, [a2], #4
        subs a3, a3, #1
        bne loop

        bx lr
```

EXAMPLE 8 (ARM ASM, RASPBERRY PI, GNU/LINUX)

```
pi@raspberrypi: 08$ ./8
Source array:
1 2 3
Destination array before copy:
67048 67008 0
Destination array after copy:
1 2 3
pi@raspberrypi: 08$ objdump -D 8
…
00010510 <main>:
   10510:       e52de004        push    {lr}            ; (str lr, [sp, #-4]!)
   10514:       e24dd024        sub     sp, sp, #36     ; 0x24
   10518:       e59f207c        ldr     r2, [pc, #124]  ; 1059c <main+0x8c>
   1051c:       e28d3010        add     r3, sp, #16
   10520:       e8920007        ldm     r2, {r0, r1, r2}
   10524:       e8830007        stm     r3, {r0, r1, r2}
   10528:       e3a03003        mov     r3, #3
   1052c:       e58d301c        str     r3, [sp, #28]
   10530:       e59f0068        ldr     r0, [pc, #104]  ; 105a0 <main+0x90>
   10534:       ebffff80        bl      1033c <puts@plt>
   10538:       e28d3010        add     r3, sp, #16
   1053c:       e59d101c        ldr     r1, [sp, #28]
   10540:       e1a00003        mov     r0, r3
```

PUSH AND SUB MOVE THE TOP OF THE STACK 40 BYTES DOWNWARDS PRESERVING 8-BYTE ALIGNMENT

SYMBOL

ADDRESS

PC=0X10530+8=0X10538
[PC, #104] = [PC, #0X68]
=0X105A0

CALL TO PRINTF() OPTIMIZED INTO A CALL TO PUTS()

PUTS() ARGUMENT CONTAINED INTO WORD AT 0X105A0

EXAMPLE 8 (ARM ASM, RASPBERRY PI, GNU/LINUX)

```
00010510 <main>:
   10510:       e52de004        push    {lr}                ; (str lr, [sp,
   10514:       e24dd024        sub     sp, sp, #36         ; 0x24
   10518:       e59f207c        ldr     r2, [pc, #124]      ; 1059c <main+0
   1051c:       e28d3010        add     r3, sp, #16
   10520:       e8920007        ldm     r2, {r0, r1, r2}
   10524:       e8830007        stm     r3, {r0, r1, r2}
   10528:       e3a03003        mov     r3, #3
   1052c:       e58d301c        str     r3, [sp, #28]
   10530:       e59f0068        ldr     r0, [pc, #104]      ; 105a0 <main+0x90>
   10534:       ebffff80        bl      1033c <puts@plt>
   10538:       e28d3010        add     r3, sp, #16
   1053c:       e59d101c        ldr     r1, [sp, #28]
   10540:       e1a00003        mov     r0, r3
…
   105a0:       00010634        andeq   r0, r1, r4, lsr r6
   105a4:       00010644        andeq   r0, r1, r4, asr #12
   105a8:       00010664        andeq   r0, r1, r4, ror #12
…
   10634:       72756f53        rsbsvc  r6, r5, #332        ; 0x1
   10638:       61206563                                    ; <UNDEFINED>
   1063c:       79617272        stmdbvc r1!, {r1, r4, r5, r6
   10640:       0000003a        andeq   r0, r0, sl, lsr r0
```

LOAD ADDRESS

VALUE LOADED INTO R0 FROM ADDRESS 0X105A0: STRING ADDRESS 0X10634

10637	10636	10635	10634
72 r	75 u	6F o	53 S
1063B	1063A	10639	10638
61 a	20	65 e	63 c
1063F	1063E	1063D	1063C
79 y	61 a	72 r	72 r
10643	10642	10641	10640
00 Padding	00 Padding	00 NUL	3A :

NUL-terminated string ("Source array:") and padding stored at 0x10634 dumped as little-endian word values