Raspberry PI ASM 32 bit

Bartolomeo Montrucchio

Politecnico di Torino

Dipartimento di Automatica e Informatica (DAUIN)

Torino - Italy

Basic example (gcc based) 1/3

```
int main()
{
        int i;
        i = i + 5;
        return 9;
}
```

```
$gcc –S –o first.s first.c

$gcc –o first first.s
$./first
```

ATTENTION: there are different syntaxes:

- one for compiling with gcc
- one for compiling with as and ld
- one if the .s file comes from gcc –S
- one for KEIL
- VERIFY the README
- please note that syntax for 64 bits is again different

Basic example (gcc based) 2/3

```
# cat first.s

.syntax unified
.arch armv7-a
.eabi_attribute 27, 3
.eabi_attribute 28, 1
.fpu vfpv3-d16
.eabi_attribute 20, 1
.eabi_attribute 21, 1
.eabi_attribute 23, 3
.eabi_attribute 24, 1
.eabi_attribute 25, 1
.eabi_attribute 26, 2
.eabi_attribute 30, 6
.eabi_attribute 18, 4
.thumb
.file           "first.c"
.text
.align          2
.global         main
.thumb
.thumb_func
.type           main, %function
main:
    @ args = 0, pretend = 0, frame = 8
    @ frame_needed = 1, uses_anonymous_args = 0
    @ link register save eliminated.
    push    {r7}
    sub     sp, sp, #12
    add     r7, sp, #0
    ldr     r3, [r7, #4]
    add     r3, r3, #5
    str     r3, [r7, #4]
    mov     r3, #9
    mov     r0, r3
    add     r7, r7, #12
    mov     sp, r7
    pop     {r7}
    bx      lr
    .size   main, .-main
    .ident  "GCC: (Debian 4.6.3-14) 4.6.3"
    .section .note.GNU-stack,"",%progbits
```

Basic example (gcc based) rewritten 3/3

```
root@debian-armhf:~/constructs_32bit# cat test32.s
@ compile with:
@   gcc -o test32  test32.s

        .global main
        .func main

main:
            ADD r0, r0, #5
_exit:
            MOV R0, #9   @ return value to the operating
                    @ system; see it with echo $?
            MOV R7, #1 @ exit through system call
            SWI 0
root@debian-armhf:~/constructs_32bit# gcc test32.s
root@debian-armhf:~/constructs_32bit# ./a.out
root@debian-armhf:~/constructs_32bit# echo $?
9
```

Different syntaxes (32 bit)

- as+ld:
  - .global _start /* to begin code */
  - _start:
  - mov r7, #1 /* to end code */
  - swi 0 /* or svc 0 */
- gcc (but not gcc –S):
  - .global main /* to begin code */
  - .func main
  - main:
  - same way of as+ld or: /* to end code */
    - mov pc, lr or:
    - bx lr
- comparison as+ld, gcc, KEIL in materiale/

gdb

- gdb is the standard debugger
- gdb file   (where file is from: as –g –o file.o file.s)
- b for break point
- r for run, c to continue after a break up to next break
- s for a step (n to skip procedures)
- i r to see registers (info registers)
- x/26db &fibonacci shows 26 unsigned bytes starting at the fibonacci's memory address; otherwise use addresses, maybe from registers
- see file in materiale/

A program in two files (p.37 B.Smith)

```
/* part1.s file                */
            .global _start
_start:
            MOV R0, #65
            BAL _part2        @ branch
always
```

```
/* part2.s file                */
            .global _part2
_part2:
            MOV R7, #1
            SWI 0
```

- as –o part1.o part1.s
- as –o part2.o part2.s
- ld –o allparts part1.o part2.o

Simple 32 bit addition (p.62)

```
/* Add two 32-bit numbers together      */
/* Perform R0=R1+R2                      */
        .global  _start
_start:
        MOV R1, #50             @ Get 50 into R1
        MOV R2, #60             @ Get 60 into R2
        ADDS  R0, R1, R2        @ Add the two, result in R0

        MOV R7, #1              @ exit through syscall
        SWI 0
```

- the result of echo $? will be 110 (50+60)

64-bit addition (p.63)

```
/* Add two 64-bit numbers together        */
          .global  _start
_start:
          MOV R2, #0xFFFFFFFF          @ low half number 1
          MOV R3, #0x1            @ hi half number 1
          MOV R4, #0xFFFFFFFF          @ low half number 2
          MOV R5, #0xFF               @ hi half number 2
          ADDS  R0, R2, R4       @ add low and set flags (S)
          ADCS  R1, R3, R5       @ add hi with carry and set
flags

          MOV R7, #1             @ exit through syscall
          SWI 0
```

- the result of echo $? will be 254 (FE, lowest byte of R0)

32-bit multiplication (p.67)

```
/* multiply two numbers R0=R1*R2       */
        .global _start
_start:
        MOV R1, #20     @ R1=20
        MOV R2, #5      @ R2=5
        MUL R0, R1, R2          @ R0=R1*R2

        MOV R7, #1      @ exit through syscall
        SWI 0
```

- the result of echo $? will be 100 (=20*5)
- only registers with MUL
- no R15 as destination
- destination register can not be used as operand (here R1)

32-bit multiplication using MLA(p.68)

```
/* multiply two numbers R0=R1*R2 and add additional  */
        .global  _start
_start:
        MOV R1, #20             @ R1=20
        MOV R2, #5              @ R2=5
        MOV R3, #10             @ R3=10
        MLA R0, R1, R2, R3      @ R0=(R1*R2)+R3

        MOV R7, #1              @ exit through syscall
        SWI 0
```

- the result of echo $? will be 110 (=R1*R2+R3)
- ARM DOES NOT provide division instructions (see later)
  - ARM 64 does provide!

Using System Call 4 to write a string on the screen (p.73)

```
/* How to use Syscall 4 to write a string */
        .global _start
_start:
        MOV R7, #4          @ Syscall number
        MOV R0, #1          @ Stdout is monitor
        MOV R2, #19         @ string is 19 chars long
        LDR R1,=string      @ string located at string:
        SWI 0
_exit:
                                    @ exit syscall
        MOV R7, #1
        SWI 0
.data
string:
.ascii "Hello World String\n"
```

- SWI is for SoftWare Interrupt

Using System Call 3 to read from keyboard (p.75)

```
/* How to use Syscall 3 to read from keyboard */
            .global _start
_start:
_read:
                                    @ read syscall
            MOV R7, #3  @ Syscall number
            MOV R0, #0  @ Stdin is keyboard
            MOV R2, #5  @ read first 5 characters
            LDR R1,=string          @ string placed at string:
            SWI 0
_write:
                        @ write syscall
            MOV R7, #4  @ Syscall number
            MOV R0, #1  @ Stdout is monitor
            MOV R2, #19             @ string is 19 chars long
            LDR R1,=string          @ string located at string:
            SWI 0
_exit:
  @ exit syscall
            MOV R7, #1
            SWI 0
.data
string:
.ascii "Hello World String\n"
```

- SWI is for SoftWare Interrupt

Converting character case (p.83)

```
/* Using ORR to toggle a character case */
            .global _start
_start:
_read:                                  @ read syscall
            MOV R7, #3  @ Syscall number
            MOV R0, #0  @ Stdin is keyboard
            MOV R2, #1  @ read one character only
            LDR R1,=string              @ string at string:
            SWI 0
_togglecase:
            LDR R1, =string                     @ address of char
            LDR R0, [R1]                @ load it into R0
            ORR R0, R0, #0x20           @ change case
            STR R0, [R1]                @ write char back
_write:                                         @ write syscall
            MOV R7, #4                  @ Syscall number
            MOV R0, #1                  @ Stdout is monitor
            MOV R2, #1                  @ string is 1 char long
            LDR R1,=string                      @ string at start:
            SWI 0
_exit:
   @ exit syscall
            MOV R7, #1
            SWI 0
.data
string:     .ascii " "
```

- SWI is for SoftWare Interrupt

Printing a number as a binary string (p.85)

```
/**** Convert number to binary for printing ****/
        .global _start
_start:
        MOV R6, #251            @ Number to print in R6
        MOV R10, #1             @ set up mask
        MOV R9, R10, LSL #31
        LDR R1, = string        @ Point R1 to string
_bits:
        TST R6, R9              @ TST no, mask
        BEQ _print0
        MOV R8, R6              @ MOV preserve, no
        MOV R0, #49             @ ASCII '1'
        STR R0, [R1]            @ store 1 in string
        BL _write               @ write to screen
        MOV R6, R8              @ MOV no, preserve
        BAL _noprint1
_print0:
        MOV R8, R6              @ MOV preserve, no
        MOV R0, #48             @ ASCII '0'
        STR R0, [R1]            @ store 0 in string
        BL _write
        MOV R6, R8              @ MOV no, preserve
_noprint1:
        MOVS R9, R9, LSR #1     @ shuffle mask bits
        BNE _bits

_exit:
        MOV R7, #1
        SWI 0

_write:
        MOV R0, #1
        MOV R2, #1
        MOV R7, #4
        SWI 0
        MOV PC, LR

.data
string:
        .ascii " "
```

Printing a number as a binary string with B (p.101)

```
/**** Convert to binary for printing ****/
        .global _start
_start:
        MOV R6, #215     @ Number to print in R6
        MOV R10, #1              @ set up mask
        MOV R9, R10, LSL #31
        LDR R1, = string       @ Point R1 to string
_bits:
        TST R6, R9               @ TST no, mask
        MOVEQ R0, #48            @ ASCII '0'
        MOVNE R0, #49            @ ASCII '1'
        STR R0, [R1]     @ store 1 in string
        MOV R8, R6       @ MOV preserve, no
        BL _write        @ write to screen
        MOV R6, R8       @ MOV no, preserve
MOVS R9, R9, LSR #1
@ shuffle mask bits
        BNE _bits
_exit:
        MOV R7, #1
        SWI 0

_write:
        MOV R0, #1
        MOV R2, #1
        MOV R7, #4
        SWI 0
        BX LR

.data
string: .ascii " "
```

If 1/2

```
# cat if.c
int main()
{
        int a;
        a = 3;
        if (a == 4)
        {
                a+= 8;
        }
        else
        {
                a+= 20;
        }
        return a;
}
```

```
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        mov     r3, #3
        str     r3, [r7, #4]
        ldr     r3, [r7, #4]
        cmp     r3, #4
        bne     .L2
        ldr     r3, [r7, #4]
        add     r3, r3, #8
        str     r3, [r7, #4]
        b       .L3
.L2:
        ldr     r3, [r7, #4]
        add     r3, r3, #20
        str     r3, [r7, #4]
.L3:
        ldr     r3, [r7, #4]
        mov     r0, r3
        add     r7, r7, #12
        mov     sp, r7
        pop     {r7}
        bx      lr
```

If 2/2

$ gcc -O3 -S -o if_O3.s if.c
NOTE: same result also with –O1 and –O2!!

```
root@debian-armhf:~/constructs_32bit# cat if_O3.s
…….
main:
	@ args = 0, pretend = 0, frame = 0
	@ frame_needed = 0, uses_anonymous_args = 0
	@ link register save eliminated.
	movs	r0, #23
	bx	lr
	.size	main, .-main
	.ident	"GCC: (Debian 4.6.3-14) 4.6.3"
	.section	.note.GNU-stack,"",%progbits
```

For

```
int main()
{
        int i;
        int j=0;

        for (i=0; i<10; i++)
        {
                j++;
        }

        return j;
}
```

If –O3 is used:
gcc -S -O3 -o for_simple_O3.s for_simple.c
This is the result:

```
        movs    r0, #10
        bx      lr
```

```
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        mov     r3, #0
        str     r3, [r7, #0]
        mov     r3, #0
        str     r3, [r7, #4]
        b       .L2
.L3:
        ldr     r3, [r7, #0]
        add     r3, r3, #1
        str     r3, [r7, #0]
        ldr     r3, [r7, #4]
        add     r3, r3, #1
        str     r3, [r7, #4]
.L2:
        ldr     r3, [r7, #4]
        cmp     r3, #9
        ble     .L3
        ldr     r3, [r7, #0]
        mov     r0, r3
        add     r7, r7, #12
        mov     sp, r7
        pop     {r7}
        bx      lr
```

While

```c
int main()
{
	int j=10;

		int i=0;
		while (i<10)
		{
				j+=3;

				i++;
		}
}
```

```asm
	push	{r7}
	sub	sp, sp, #12
	add	r7, sp, #0
	mov	r3, #10
	str	r3, [r7, #4]
	mov	r3, #0
	str	r3, [r7, #0]
	b	.L2
.L3:
	ldr	r3, [r7, #4]
	add	r3, r3, #3
	str	r3, [r7, #4]
	ldr	r3, [r7, #0]
	add	r3, r3, #1
	str	r3, [r7, #0]
.L2:
	ldr	r3, [r7, #0]
	cmp	r3, #9
	ble	.L3
	mov	r0, r3
	add	r7, r7, #12
	mov	sp, r7
	pop	{r7}
	bx	lr
	bx	lr
```

Multiplication

```
int main()
{
		long a=7;
// long is 32 bits on ARM v7
// and 64 bits on ARM v8
		long long b=12;
// long long is always 64 bits

		a *= a;
		b *= b;
}
```

```
push    {r7}
sub     sp, sp, #20
add     r7, sp, #0
mov     r3, #7
str     r3, [r7, #12]
mov     r2, #12
mov     r3, #0
strd    r2, [r7] #0]
ldr     r3, [r7, #12]
ldr     r2, [r7, #12]
mul     r3, r2, r3
str     r3, [r7, #12]
ldr     r3, [r7, #4]
ldr     r2, [r7, #0]
mul     r2, r2, r3
ldr     r3, [r7, #4]
ldr     r1, [r7, #0]
```

```
mul     r3, r1, r3
adds    r1, r2, r3
ldr     r2, [r7,

ldr     r3, [r7,

umull   r2, r3, r2,

adds    r1, r1, r3
mov     r3, r1
strd    r2, [r7]
strd    r2, [r7]
mov     r0, r3
add     r7, r7,
        #20
```

Division

```c
int main()
{
        long a=42;
        long b=7;
        long c=0;

    c=a/b;
        return c;
}
```

```
push    {r7, lr}
sub     sp, sp, #16
add     r7, sp, #0
mov     r3, #42
str     r3, [r7, #12]
mov     r3, #7
str     r3, [r7, #8]
mov     r3, #0
str     r3, [r7, #4]
ldr     r0, [r7, #12]
ldr     r1, [r7, #8]
bl      __aeabi_idiv
mov     r3, r0
str     r3, [r7, #4]
ldr     r3, [r7, #4]
mov     r0, r3
add     r7, r7, #16
mov     sp, r7
pop     {r7, pc}
```

Load from a memory variable

```
/* copyright Bernat Rafales with modifications*/
/* as -o memload.o memload.s  ld -o memload memload.o */
.data
.balign 4 /* Ensure variable is 4-byte aligned */
myvar1:  /* Define storage for myvar1 */
.word 3  /* Contents of myvar1 is just 4 bytes containing value '3' */
.balign 4  /* Ensure variable is 4-byte aligned */
myvar2:  /* Define storage for myvar2 */
.word 4 /* Contents of myvar2 is just 4 bytes containing value '4' */
/* -- Code section */
.text
.balign 4  /* Ensure code is 4 byte aligned */
.global _start
_start:
   ldr r1, addr_of_myvar1 /* r1 ← &myvar1 */
   ldr r1, [r1]          /* r1 ← *r1 */
   ldr r2, addr_of_myvar2 /* r2 ← &myvar2 */
   ldr r2, [r2]          /* r2 ← *r2 */
   add r0, r1, r2        /* r0 ← r1 + r2 */
   mov r7, #1
   swi 0
/* Labels needed to access data */
addr_of_myvar1 : .word myvar1
addr_of_myvar2 : .word myvar2
```

Very important!
In 386 and x86-64 architectures, instructions can access registers or memory, so we could add two numbers, one of which is in memory!
Here it is NOT possible.
And you have to pass through the relocation address (known after linking) put in code segment, in order to modify the actual variable, put in the data segment

Write in a memory variable

```
.data
.balign 4
myvar1:
  .word 0   /* Contents of myvar1 is just '3' */
.balign 4
myvar2:
  .word 0  /* Contents of myvar2 is just '3' */
.text
.balign 4
.global _start
_start:
   ldr r1, addr_of_myvar1 /* r1 ← &myvar1 */
   mov r3, #3             /* r3 ← 3 */
   str r3, [r1]           /* *r1 ← r3 */
   ldr r2, addr_of_myvar2 /* r2 ← &myvar2 */
   mov r3, #4             /* r3 ← 4 */
   str r3, [r2]           /* *r2 ← r3 */
   /* Same instructions as above */
   ldr r1, addr_of_myvar1 /* r1 ← &myvar1 */
   ldr r1, [r1]           /* r1 ← *r1 */
   ldr r2, addr_of_myvar2 /* r2 ← &myvar2 */
   ldr r2, [r2]           /* r2 ← *r2 */
   add r0, r1, r2
   mov r7, #1
   svc 0
```

```
/* Labels needed to access data */
addr_of_myvar1 : .word myvar1
addr_of_myvar2 : .word myvar2
```

Very important again!
In 386 and x86-64 architectures, instructions can access registers or memory, so we could add two numbers, one of which is in memory!
Here it is NOT possible.
And you have to pass through the relocation address (known after linking) put in code segment, in order to modify the actual variable, put in the data segment