F28HS Hardware-Software Interface: Systems Programming

Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh

Semester 2 — 2022/23

[0]No proprietary software has been used in producing these slides

Outline

Tutorial 4: Inline Assembler with gcc

- So far we have developed either C or Assembler programs separately.
- Linking the compiled code of both C and Assembler sources together we can call one from the other.
- This is ok, but sometimes inconvenient because
  - errors occur only at link time, and carry little information
  - we can't easily parameterise the Assembler code (e.g. with the `gpio` base address)
- In this tutorial we will cover **how to embed assembler code into a C program, using the `gcc` and the GNU toolchain**

Basic ARM Assembler Instructions

`MOV R0, R1`	move the value from register `R1` into register `R0`
`LDR R0, [R1]`	load the value from the location stored in register `R1` into register `R0`
`STR R0, [R1]`	store the value in register `R0` into the location stored in register `R1`
`ADD R0, R1, R2`	add the values in registers `R1` and `R2`, and store the result in register `R0`

A Simple Example

Look-up the value in `val` and copy it to `val3`:

Essentials

`val` provides the input
`asm` code returns its value
`val3` receives the output

```
static volatile int val = 1024, val3;
asm(/* multi-line example of value look-up and return
    */
    "\tMOV␣R0,␣%[value]\n"          /* load the address
       into R0 */
    "\tLDR␣%[result],␣[R0,␣#0]\n"   /* get and return
       the value at that address */
    : [result] "=r" (val3)    /* output parameter */
    : [value] "r" (&val)      /* input parameter */
    : "r0", "cc" );           /* registers used */

fprintf(stderr, "Value␣lookup␣at␣address␣%x␣(expect␣%d)
   :␣%d\n", &val, val, val3);
```

[0]Sample source in sample0.c; see also ARM inline assembly blog

Example explained

- The `asm` command defines a block of assembler code that is put at that location into the C code (embedded).
- The assembler code itself is written as a sequence of strings, each starting with a TAB (`\t`) and ending with a newline (`\n`) to match usual assembler code formatting.
- Inside the strings, the code can refer to arguments provided in the "output parameter" and "input parameter" sections.
- These sections define a **name** (e.g. `result`) that can be used in the assembler code (e.g. `%[result]`), and which is bound to a concrete variable or value (e.g. `val3`).
- Think of these in the same way as formatting strings in `printf` statements.

Example explained (cont'd)

- For example the line
  `: [result] "=r" (val3)`
  says "the name `result`, which is referred to in the assembler code as `%[result]`, is bound to the C variable `val3`; moreover, it should be represented as a register (`"r"`)"
- So, what this example code does is to load the address of the C variable `val` into the register `R0`, and then to load the value at this address, i.e. the contents of the C variable `val`, into the C variable `val3`, which should be kept in a register (`"r"`)
- The last section of the `asm` block defines which registers are modified by this assembler block. This information is needed by the compiler when doing register allocation.

GCC Extended Assembler Commands

Using gcc you can embed assembler code into your C programs, i.e. write "inline assembler" code in C.
The format for the inline assembler code is

```
asm [volatile] ( AssemblerTemplate
                 : OutputOperands
               [ : InputOperands
               [ : Clobbers ] ])
```

`AssemblerTemplate`: This is a literal string that is the template for the assembler code. It is a combination of fixed text and tokens that refer to the input, output, and goto parameters. `OutputOperands`: A comma-separated list of the C variables modified by the instructions in the AssemblerTemplate. An empty list is permitted.
`InputOperands`: A comma-separated list of C expressions read by the instructions in the AssemblerTemplate. An empty list is permitted.
`Clobbers`: A comma-separated list of registers or other values changed by the AssemblerTemplate, beyond those listed as outputs.
An empty list is permitted.

[0]See GCC Manual, Section "Extended Asm"

Another Example

Using a pair data structure, the function below computes the sum of both fields.

```
typedef struct {
  ulong min;  ulong max;
} pair_t;

ulong sumpair_asm(pair_t *pair) {
  ulong res;
  asm volatile(/* sum over int values */
               "\tLDR␣R0,␣[%[inp],␣#0]\n"
               "\tLDR␣R1,␣[%[inp],␣#4]\n"
               "\tADD␣R0,␣R0,␣R1\n"
               "\tMOV␣%[result],␣R0\n"
               : [result] "=r" (res)
               : [inp] "r" (pair)
               : "r0", "r1", "cc"  );
  return res;
}
```

Essentials

C variable `pair` is passed as `inp`
`"r"`: keep in register
`"=r"`: the register is written to

Modifiers and constraints to the input/output operands

When mapping **names** to C **variables** or **expressions**, the following constraints and modifiers can be specified:

Constraint	Specification
`f`	Floating point registers f0 ... f7
`r`	General register r0 ... r15
`m`	Memory address
`I`	Immediate value

Modifier	Specification
`=`	Write-only operand, usually used for all output operands
`+`	Read-write operand, must be listed as an output operand
`&`	A register that should be used for output only

E.g. `: [result] "=r" (res)`
means that the name `result` should be a register in the assembler code, and that it will be written to, by the assembler code.

Extended inline assembler: Example

Using a pair data structure, the function below puts the smaller value into the `min` and the larger value into the `max` field:

```
typedef struct {
  ulong min;  ulong max;
} pair_t;

void minmax_c(pair_t *pair) {
  ulong t;
  if (pair->min > pair->max) {
    t = pair->min;
    pair->min = pair->max;
    pair->max = t;
  }
}
```

[0] Sample source: sumav1_asm.c

Extended inline assembler: Example

```
void minmax_asm(pair_t *pair) {
  pair_t *res;
  asm volatile("\tLDR␣R0,␣[%[inp],␣#0]\n"
               "\tLDR␣R1,␣[%[inp],␣#4]\n"
               "\tCMP␣R0,␣R1\n"
               "\tBLE␣done\n"
               "\tMOV␣R3,␣R0\n"
               "\tMOV␣R0,␣R1\n"
               "\tMOV␣R1,␣R3\n"
               "done:␣STR␣R0,␣[%[inp],␣#0]\n"
               "\tSTR␣R1,␣[%[inp],␣#4]\n"
               : [result] "=r" (res)
               : [inp] "r" (pair)
               : "r0", "r1", "r3", "cc"  );
}
```

Discussion

- `inp` needs to be in a register, because it contains the base address in a load operation (`LDR`)
- we don't use `res` in this case, but it usually needs the `"=r"` modifier and constraint
- the clobber list must name **all registers that are modified** in the code: `r0, r1, r3`
- we could pass in an immediate value `sizeof(ulong)` and use it instead of the literal `#4` to make the code less hardware-dependent

Summary

- With gcc's in-line assembler commands (`asm`) you can embed assembler code into C code.
- This avoids having to write code in separate files and then link them together.
- The assembler code can be parameterised over C variables and expressions, to simplify passing arguments.
- Care needs to be taken to define **constraints** and **modifiers** (keep data in registers or memory)
- Registers that are modified need to be explicitly identified in the "clobber list".
- It is recommended to use such in-line assembler code for CW2, where you need to develop an applicaion in C and assembler.

Sample sources: sample0.c, and sumav1_asm.c and Gitlab repo Inline Assembler