Lancaster University

Inline Assembler

The beauty of executing assembler routines within C using GCC

Lecture goals

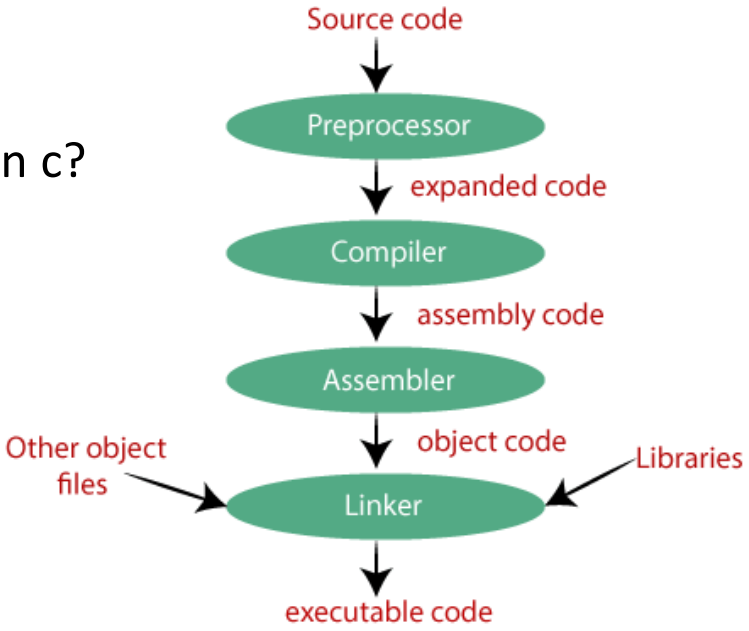
Lancaster University

- Code disassemble.
- Learn how to inline assembly from C.
  - Compiler assembly vs hand-written assembly.
- Assembly examples.

C and Assembly

Lancaster University

- GCC translates C into machine code
- Why would we want to write assembler within c?
- We introduced a compiler to avoid this!
- Some common reasons
  - Outsmart gcc logic
  - Specific optimisations
  - Close to the hardware programming (drivers)
  - Embed existing code fragments
  - Performance



Compilation example

Lancaster University

```
int main() {
      uBit.init();
      int s;
      int a = microbit_random(10);
      int b = microbit_random(10);
      s = a + b;
      uBit.serial.printf("Result is %d\n", s);
      return 0;
}
```

Disassemble example

Lancaster University

```
push {r4, r5, lr}
ldr r5, [pc, #40] @ (0x1c600 <main()+44>)
mov r0, r5
bl 0x213a0 <codal::MicroBit::init()>
movs r0, #10
bl 0x1d028 <codal::microbit_random(int)>
mov r4, r0
movs r0, #10
bl 0x1d028 <codal::microbit_random(int)>
mov r2, r0
ldr r1, [pc, #20] @ (0x1c604 <main()+48>)
addw r0, r5, #1652 @ 0x674
add r2, r4
bl 0x24744 <codal::Serial::printf(char const*, ...)>
movs r0, #0
pop {r4, r5, pc}
```

Save value of preserved registers.

Disassemble example

Lancaster University

```
push {r4, r5, lr}
ldr r5, [pc, #40] @ (0x1c600 <main()+44>)
mov r0, r5
bl 0x213a0 <codal::MicroBit::init()>
movs r0, #10
bl 0x1d028 <codal::microbit_random(int)>
mov r4, r0
movs r0, #10
bl 0x1d028 <codal::microbit_random(int)>
mov r2, r0
ldr r1, [pc, #20] @ (0x1c604 <main()+48>)
addw r0, r5, #1652 @ 0x674
add r2, r4
bl 0x24744 <codal::Serial::printf(char const*, ...)>
movs r0, #0
pop {r4, r5, pc}
```

Get the address of the uBit object.

Call uBit init function.

Disassemble example

Lancaster University

Variable	Register
a	r4
b	r2
s	Optimized out

```
push {r4, r5, lr}
ldr r5, [pc, #40] @ (0x1c600 <main()+44>)
mov r0, r5
bl 0x213a0 <codal::MicroBit::init()>
movs r0, #10
bl 0x1d028 <codal::microbit_random(int)>
mov r4, r0
movs r0, #10
bl 0x1d028 <codal::microbit_random(int)>
mov r2, r0
ldr r1, [pc, #20] @ (0x1c604 <main()+48>)
addw r0, r5, #1652 @ 0x674
add r2, r4
bl 0x24744 <codal::Serial::printf(char const*, ...)>
movs r0, #0
pop {r4, r5, pc}
```

Init r0 to 10.
Call microbit_random(10).
Result in register r0.

Disassemble example

Lancaster University

Printf argument	registers
uBit.serial addr	r0
String addr	r1
s	r2

```
push {r4, r5, lr}
ldr r5, [pc, #40] @ (0x1c600 <main()+44>)
mov r0, r5
bl 0x213a0 <codal::MicroBit::init()>
movs r0, #10
bl 0x1d028 <codal::microbit_random(int)>
mov r4, r0
movs r0, #10
bl 0x1d028 <codal::microbit_random(int)>
mov r2, r0
ldr r1, [pc, #20] @ (0x1c604 <main()+48>)
addw r0, r5, #1652 @ 0x674
add r2, r4
bl 0x24744 <codal::Serial::printf(char const*, ...)>
movs r0, #0
pop {r4, r5, pc}
```

Call printf.

Inline Assembly

Lancaster University

- GCC allows **inline** assembler
- The compiler inserts assembler code in the code of its caller
- This removes function call overheads (recall, function prolog)
- The key word asm or __asm__ is used to include code:

```
asm("assembly code");
```

- Assembly is architecture-specific.
  - You ARM assembly will not be transcoded into x86.
- You can also compile an assembly file into an obj file.

Basic Inline Assembler

Lancaster University

- Syntax
  - `asm("assembly code");`
  - `__asm__ ("assembly code");`
  - The `__asm__` variant is used to avoid conflicts
- Example
  - `asm("mov r0, r1");`
  - Copy the contents register `r1` to register `r0`

Extended Inline Assembler

Lancaster University

- The embedded assembler code must interact with c code
  - Parameters must be passed on to the assembler program
  - Results must be passed on to C code
- Syntax:

```
asm ( "assembly code"
: output operands                    /* optional */
: input operands                     /* optional */
: list of clobbered registers        /* optional */
);
```

Input/output Operands

Lancaster University

```
int a = 100, b;
asm ("movl r0, %[a];"
     "movl %0, r0;"
   : "=r" ( b )       /* output */
   : [a] "r"  ( a )   /* input */
   : ”r0"             /* clobbered register */
    );
```

Constraint	Usage in ARM state
I	Immediate value, e.g. ORR R0, R0, #op
J	Indexing constants -4095 .. 4095, e.g. LDR R1, [PC, #op]
M	Constant in the range of 0 .. 32. e.g. MOV R2, R1, ROR #op
m	Any valid memory address
r	General register r0 .. R15, e.g. SUB op1, op2, op3
X	Any operand

Input/output Operands - Example

Lancaster University

```
int a = 100, b ;
asm ("movl r0, %[a];"
     "movl %[b], r0;"
   : [b] "=r" ( b )     /* output */
   : [a] "r"  ( a )     /* input */
   : ”r0"               /* clobbered register */
    );
```

- b is the output operand, referred to by %[b]
- a is the input operand referred to by %[a]
- register r0 should not be used to store values (used intern)

A Simple Example

Lancaster University

- Replace addition with assembler code
- %0 as input for a and as well as output for result s
- variable b is represented by %1

```
#include <stdio.h>

int main(){
    int a=1;
    int b=2;
    int s;
    //s = a+b;
    __asm__ ( "add %0, %1;"
            : "=r" (s)
            : ”0" (a), ”r" (b) );
    printf("sum = %d\n", s);
}
```

Example Code Output

Lancaster University

```
arm-none-eabi-objdump -S build/main.obj
```

- Output (gcc disass) using c variant:

```
<+2>: movs r0, #10
<+4>: bl 0x1d01c <codal::microbit_random(int)>
<+8>: mov r4, r0
<+10>: movs r0, #10
<+12>: bl 0x1d01c <codal::microbit_random(int)>
<+14>: mov r2, r4
<+18>: add r2, r0
<+20>: add r2. r0
<+22> r2, r0 ldr r1, [pc, #12] @ (0x1c5f4 <main()+32>)
<+24>: ldr r0, [pc, #12] @ (0x1c5f8 <main()+36>)
```

- Output (gcc disass) using assembler variant:

```
<+2>: movs r0, #10
<+4>: bl 0x1d020 <codal::microbit_random(int)>
<+8>: mov r4, r0
<+10>: movs r0, #10
<+12>: bl 0x1d020 <codal::microbit_random(int)>
<+16>: ldr r1, [pc, #16] @ (0x1c5f8 <main()+36>)
<+18>: mov r3, r0
<+20>: mov r2, r4
<+22>: ldr r0, [pc, #16] @ (0x1c5fc <main()+40>)
<+24>: add r2, r3
```

You do not need this move

Register Allocation

Lancaster University

- CPUs have a limited register count.
  - Use main memory to store data.
  - Copy data into registers to process them.

- Register Allocation: a compiler must assign variables into processor register.
  - Any two variable must not be assigned to the same register at any point.
  - Use Spilling to store variable values.
  - Coalescing will aim to optimize register allocation to reduce value copying.
  - Graph coloring problem and liveness analysis.

Volatile

Lancaster University

- Sometimes it is important to execute assembler code exactly where we put it (e.g. sequence of register usage is important)
- GCC may move code for optimisation purpose
- Use the volatile qualifier for instructions with processor side-effects.
  - Disable compiler optimizations, which might lead code block removal (compiling with flag -O1 or higher).

```
volatile asm("assemler");
__volatile__ __asm__("assembler");
```

Example Code Output

Lancaster University

- Output (gcc disass) using volatile inline asm:

```
<+2>: movs r0, #10
<+4>: bl 0x1d01c <codal::microbit_random(int)>
<+8>: mov r4, r0
<+10>: movs r0, #10
<+12>: bl 0x1d01c <codal::microbit_random(int)>
<+16>: mov r2, r0
<+18>: add r2, r4
<+20>: ldr r1, [pc, #12] @ (0x1c5f4 <main()+32>)
<+22>: ldr r0, [pc, #12] @ (0x1c5f8 <main()+36>))
```

- Output (gcc disass) using inline asm:

```
<+2>: movs r0, #10
<+4>: bl 0x1d020 <codal::microbit_random(int)>
<+8>: mov r4, r0
<+10>: movs r0, #10
<+12>: bl 0x1d020 <codal::microbit_random(int)>
<+16>: ldr r1, [pc, #16] @ (0x1c5f8 <main()+36>)
<+18>: mov r3, r0
<+20>: mov r2, r4
<+22>: ldr r0, [pc, #16] @ (0x1c5fc <main()+40>)
<+24>: add r2, r3
```

```
asm volatile( "add %0, %1, %2;"
            : "=r" (s)
            : "0" (a), "r" (b) );
```

clobbered register

Lancaster University

- The clobbered register r0 after the third colon tells GCC that the value of r0 is to be modified inside "asm", so GCC won't use this register to store any other value.

```
#include "stdio.h"
int main(){
    int a=10, b;
    asm ("mov r0, %1;
          mov %0, r0;"
         :"=r"(b)        /* output */
         :"r"(a)         /* input */
         :"r0"          /* clobbered register */
         );
}
```

clobbered register

- The clobbered register r0 after the third colon tells GCC that the value of r0 is to be modified inside "asm", so GCC won't use this register to store any other value.
- the list can also contain special arguments:
  - "cc": The instruction modifies the condition code flags (save psr).
  - "memory": The instruction accesses unknown memory addresses.

```
#include "stdio.h"


int main(){
    int a=10, b;
    asm ("mov %a, r0;
          mov r0, %b;"
         :[b] "=r"(b)   /* output */
         :[a] "r"(a)    /* input */
         :"r0"          /* clobber reg */
         );
}
```

Usage Example (1) - MACRO

Lancaster University

- Convert a long value between little/big endian:

```
#define BYTESWAP(val) \
__asm__ __volatile__ ( \
"eor r3, %1, %1, ror #16\n\t" \
"bic r3, r3, #0x00FF0000\n\t" \
"mov %0, %1, ror #8\n\t" \
"eor %0, %0, r3, lsr #8" \
: "=r" (val) \
: "0"(val) \
: "r3", "cc" \
);
```

Usage Example (2) - Interrupts

Lancaster University

- Avoid interrupts around critical code:

```
// Disable interrupts
asm volatile("mrs r12, psr\n\t"
"orr r12, r12, #0xC0\n\t"
"msr psr, r12\n\t" : "=X" (b) :: "r12", "cc");
b *= a; /* This is safe. */
asm volatile("mrs r12, psr\n"
"bic r12, r12, #0xC0\n"
"msr psr, r12" :: "X" (b) : "r12", "cc");
```

Summary

Lancaster University

Inline Assembly

Next

Machine Code and Thumb instruction