The most successful programmable component on silicon is the microprocessor.

Fueled by a well-balanced mix of efficient implementations, flexibility, and tool support, microprocessors have grown into a key component for electronic design.

The topic of microprocessors is a very broad one; entire books are devoted to its discussion.

Our focus is to investigate the relationship between a C program and the execution of that C program on a microprocessor, in particular, on the RISC microprocessor. This will establish the cost of the C program in terms of memory footprint and execution time.

The chapter covers four different aspects of C program execution on RISC processors:

- We discuss the major architecture elements of a RISC processor, and their role in C program execution.
- We discuss the path from C programs to assembly programs to machine instructions.
- We discuss the runtime organization of a C program at the level of the machine.
General Purpose Embedded Cores

- We discuss techniques to evaluate the quality of generated assembly code, and thus evaluate the quality of the C compiler.

Processors

The most successful programmable component of the past decades is, without doubt, the microprocessor.

Just about any electronic device more complicated than a pushbutton seems to contain a microprocessor.

There have been a number of drivers for the popularity of the microprocessor:

- Microprocessors, or the stored-program concept in general, separate software from hardware through the definition of an instruction-set.
  
  No other hardware development technique has ever been able to decouple hardware and software in a similar way.

  For example, micro-programs are really shorthand notations for the control specification of a specialized datapath.
Processors

• Microprocessors come with tools (compilers and assemblers), that help a designer create applications
  The availability of a compiler to *automatically translate* a code into a binary for a microprocessor is an *enormous* advantage for development

  An *embedded software designer* can therefore be proficient in one programming language like C, and this alone allows him to move *seamlessly* across different microprocessor architectures

• There have been very few devices that have been able to cope as efficiently with *reuse* as microprocessors have done
  A general-purpose embedded core by itself is an excellent example of reuse

  Moreover, microprocessors have also *dictated the rules of reuse* for electronic system design in general
  They have defined *bus protocols* that enabled the integration of an entire system
  Their compilers have enabled the development of *standard software libs*
Processors

- No other programmable components have the same scalability as microprocessors. The same concept (i.e. stored-program computer) has been implemented across a large range of word-lengths (4-bit ... 64-bit) and basic architecture-types.

In addition, microprocessors have also extended their reach to entire chips, containing many other components, while staying ’in command’ of the system.

This approach is commonly called System-On-Chip (SoC).

The Toolchain of a Typical Microprocessor

The following figure illustrates a typical design flow to convert software source code into instructions for a processor.

A compiler or an assembler is used to convert source code into object code. Each object code file contains a binary representation of the instructions and data constants corresponding to the source code.

Plus supporting information to organize these instructions/constants in memory.
The Toolchain of a Typical Microprocessor

Fig. 6.1 Standard design flow of software source code to processor instruction
**The Toolchain of a Typical Microprocessor**

A *linker* is used to combine several object code files into a single, stand-alone executable file.

A linker will **resolve all unknown elements**, such as external data or library routines, while creating the executable file.

A *loader* program determines how the information in an executable file is organized into memory locations.

Typically, a part of the memory space is reserved for instructions, another part for constant data, another part for global data with read/write access, and so on.

A very simple microprocessor system contains at least two elements: the processor, and a memory holding instructions for the processor.

- The memory is initialized with processor instructions by the loader.

- The processor will then fetch these instructions from memory and execute them on the processor datapath.
The Toolchain of a Typical Microprocessor

Consider the following C program:

```c
1 int gcd(int a[5], int b[5]) {
2   int i, m, n, max;
3   max = 0;
4   for (i=0; i<5; i++) {
5     m = a[i];
6     n = b[i];
7     while (m != n) {
8       if (m > n)
9         m = m - n;
10      else
11         n = n - m;
12    }
13    if (max > m)
14      max = m;
15  }
16  return max;
17 }
```
From C to Assembly Instructions

18
19 int a[] = {26, 3, 33, 56, 11};
20 int b[] = {87, 12, 23, 45, 17};
21
22 int main() {
23 return gcd(a, b);
24 }

This C program that evaluates the largest among the common divisors of 5 pairs of numbers

We will inspect the C program at two lower levels of abstraction
• At the level of assembly code generated by the compiler
• At the level of the machine code stored in the executable generated by the linker

We consider embedded microprocessor architectures such as ARM or Microblaze

We will use of a **cross-compiler** to generate the executable for these microprocessors, i.e., for an ARM processor using a PC workstation
From C to Assembly Instructions

A cross-compiler generates an executable for a processor different than the machine used to run the compiler.

We use the GNU compiler toolchain to generate an ARM assembly listing is as follows:

```
> arm-linux-gcc -c -S -O2 gcd.c -o gcd.s
```

The command to generate the ARM ELF executable is as follows.

```
> /usr/local/arm/bin/arm-linux-gcc -O2 gcd.c -o gcd
```

Assembly dump of the GCD program:

```
1  gcd:
2    str   lr, [sp, #-4]!
3    mov   lr, #0
4    mov   ip, lr
5  .L13:
6    ldr   r3, [r0, ip, asl #2]
7    ldr   r2, [r1, ip, asl #2]
```
From C to Assembly Instructions

8     cmp     r3, r2
9     beq     .L17

10   .L11:
11     cmp     r3, r2
12     rsbgt    r3, r2, r3
13     rsble    r2, r3, r2
14     cmp     r3, r2
15     bne     .L11

16   .L17:
17     add     ip, ip, #1
18     cmp     lr, r3
19     movge    lr, r3
20     cmp     ip, #4
21     movgt    r0, lr
22     ldrgt    pc, [sp], #4
23     b       .L13
From C to Assembly Instructions

24  a:
25       .word  26, 3, 33, 56, 11
26  b:
27       .word  87, 12, 23, 45, 17
28 main:
29       str  lr,  [sp, #-4]!
30       ldr  r0, .L19
31       ldr  r1, .L19+4
32       ldr  lr,  [sp], #4
33       b    gcd
34       .align 2
35 .L19:
36       .word  a
37       .word  b

Use `man gcc` or `gcc --help` on the command line will list and clarify the available command-line options
From C to Assembly Instructions

Comparing the assembly program to the C program will help you to understand low-level implementation details of the C program

• Program Structure
  The overall structure of the assembly program preserves the structure of the C program

  The gcd function is on lines 1-23, the main function is on lines 28-34

  The loop structure of the C program can be identified by inspection
    In the gcd function, the inner for loop is on lines 10-15, and the outer while loop is on line 5-23

• Storage
  The constant arrays $a$ and $b$ are directly encoded as constants on lines 24-27

  The assembly code uses pointers to these arrays
    For example, the storage location at label .L19 will hold a pointer to array $a$
Function Calls

Function calls in assembly code need to handle the same semantics as a C function call.

The function call $gcd(a,b)$ has two parameters which need to be passed from \textit{main} to $gcd$.

Lines 30-32 show how the C function call is implemented.

The assembly program copies the address of these arrays into $r0$ and $r1$.

The assembly version of $gcd$ can make use of $r0$ and $r1$ as a pointer to array $a$ and $b$ respectively.

Assembly is the starting point to study the implementation details of software on a micro-processor.

Later, we will also discuss other implementation issues, such as handling of local variables, data types, memory allocation, and compiler optimizations.
From C to Assembly Instructions

The micro-processor works with **object code**, which are binary opcodes generated from assembly programs.

Compiler tools can re-create (de-assemble) the assembly code from the executable:

```
> /usr/local/arm/bin/arm-linux-objdump -d gcd
```

```
Disassembly of section .text:

00008380 <gcd>:

  8380:   e52de004  str  lr, [sp, -#4]!
  8384:   e3a0e000  mov  lr, #0   ; 0x0
  8388:   e1a0c00e  mov  ip, lr
  838c:   e790310c  ldr  r3, [r0, ip, lsl #2]
  8390:   e791210c  ldr  r2, [r1, ip, lsl #2]
  8394:   e1530002  cmp  r3, r2
  8398:   0a000004  beq  83b0 <gcd+0x30>
  839c:   e1530002  cmp  r3, r2
  83a0:   c0623003  rsbgt r3, r2, r3
```
From C to Assembly Instructions

13  83a4:   d0632002    rsble r2, r3, r2
14  83a8:   e1530002    cmp  r3, r2
15  83ac:   1afffffffa  bne  839c <gcd+0x1c>
16  83b0:   e28cc001    add  ip, ip, #1 ;0x1
17  83b4:   e15e0003    cmp  lr, r3
18  83b8:   a1a0e003    movge lr, r3
19  83bc:   e35c0004    cmp  ip, #4     ;0x4
20  83c0:   cl1a0000e    movgt r0, lr
21  83c4:   c49df004    ldrgt pc, [sp], #4
22  83c8:   eaafffffe    b  838c <gcd+0xc>
23  000083cc <main>:  
24  83cc:   e52de004    str  lr, [sp, -#4]!
25  83d0:   e59f0008    ldr  r0, [pc, #8]  
26  83d4:   e59f1008    ldr  r1, [pc, #8]  
27  83d8:   e49de004    ldr  lr, [sp], #4 
28  83dc:   eaafffffe7    b  8380 <gcd>
From C to Assembly Instructions

<table>
<thead>
<tr>
<th>Line</th>
<th>Address</th>
<th>Binary</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>83e0:</td>
<td>00010444</td>
<td><code>andeq r0, r1, r4, asr #8</code></td>
</tr>
<tr>
<td>30</td>
<td>83e4:</td>
<td>00010458</td>
<td><code>andeq r0, r1, r8, asr r4</code></td>
</tr>
</tbody>
</table>

The instructions are mapped to sections of memory, and the `.text section` holds the instructions of the program.

Each function as a particular starting address, measured relative to the beginning of the executable.

For example, the `gcd` function starts at `0x8380` and the main functions starts at `0x83cc`.

In addition to the opcode, the listing shows the binary representation of instructions.

As part of generating the executable, the `address value` of each label is added to each instruction.

For example, the `b .L13` instruction on line 23 of the original assembly is encoded as a branch to address `0x838c` on line 22 of the de-assembled code.
Simulating a C program Executing on a Microprocessor

Here, we briefly explain how to simulate an embedded micro-processor on a standard workstation.

Such simulations are very common in hardware-software codesign:
They are used to test the executables created with a cross-compiler, and to evaluate the performance of the resulting program.

Micro-processors such as ARM can be simulated with an instruction-set simulator.

The GEZEL cosimulation environment integrates several instruction-simulation engines, including:
- ARM processor (SimIt-ARM was developed by Wei Qin)
- 8051 micro-controller (Dalton 8051 developed by the team of Frank Vahid)
- picoblaze micro-controller (Picoblaze simulator was developed by Mark Six)

These simulation engines are open-source software projects.
Simulating a C program Executing on a Microprocessor

The figure below shows how instruction-set simulators are integrated into the GEZEL cosimulation engine, \textit{gplatform}

![Diagram of simulation process](image)

\textbf{Fig. 6.2} An instruction-set simulator, integrated into GEZEL, can simulate cross-compiled executables

The software part of the application is written in C, and compiled into executable format using a cross compiler
Simulating a C program Executing on a Microprocessor

The hardware part is written in GEZEL, and it specifies the platform architecture, i.e., the microprocessor, and its interaction with other hardware modules.

The combination of the GEZEL program and the cross-compiled executable format is used in a cosimulation.

All the instruction-set simulation engines in GEZEL are cycle-accurate simulators. They reflect the behavior of a processor clock-cycle by clock-cycle.

Instruction-set simulation engines can also be instruction-accurate. They run faster than cycle-accurate simulation engines b/c they handle less detail.

Here is a GEZEL program that simulates a stand-alone ARM core that executes the gcd program given earlier:

```gezel
1 ipblock myarm {
2   iptype "armsystem";
3   ipparm "exec = gcd";
```
Simulating a C program Executing on a Microprocessor

```gezel
4 } 5 6 
6 dp top { 
7   use myarm; 
8 }
9
10 system S { 
11   top; 
12 }
```

Lines 1-4 define an ARM core which runs an executable program called \textit{gcd}

The \textit{ipblock} is a special type of GEZEL module which represents a \textbf{black-box} simulation model (a simulation model without internal details)

It does not have any input/output ports (I/O ports will be introduced later)

Lines 6-12 of the GEZEL program configure the \textit{myarm} module for execution
Simulating a C program Executing on a Microprocessor

To simulate the program, we need to *cross-compile* the C application software for the ARM instruction-set simulator.

To generate output through the co-simulation, the main function of the C program is modified as follows:

```c
int main() {
    printf("gcd(a, b)= %d\n", gcd(a,b));
    return 0;
}
```

The compilation and co-simulation is now done through the following commands:

```
> /usr/local/arm/bin/arm-linux-gcc -static gcd.c -o gcd
> gplatform top.fd1core myarmarmsystem: loading executable [gcd]
gcd(a,b)=3
Total Cycles: 14338
```

The output of the simulation shows that the program takes 14338 cycles to execute.