# SOFTWARE SYNTHESIS FOR EMBEDDED PROCESSORS

Lochana Abhayawardana

lochana.abhayawardana@stud.hshl.de

# MOTIVATION

# AGENDA

1. INTRODUCTION TO SOFTWARE SYNTHESIS FOR EMBEDDED SYSTEMS
2. APPROACHES TO SOFTWARE SYNTHESIS
3. PROGRAMMING IN HIGH LEVEL-LANGUAGES
4. COMPILERS
5. INSTRUCTION SET ARCHITECTURE (ISA)
6. Reduced Instruction Set Computer (RISC)
7. ARM CPU ARCHITECTURE
8. CONCLUSION

# INTRODUCTION

- Embedded system in now days

- Software Synthesis – a critical role in reducing the design complexity

- Different approaches of the software synthesis
    1. High level synthesis
    2. Register Transfer level synthesis
    3. Instruction set synthesis

# INTRODUCTION TO SOFTWARE SYNTHESIS FOR EMBEDDED SYSTEMS

- Software synthesis
- System synthesis
- Each part below System model uses it as an input and generates an implementation for each Hardware, interface and software synthesis.
- The Software Synthesis produces binary code for programmable processing elements.
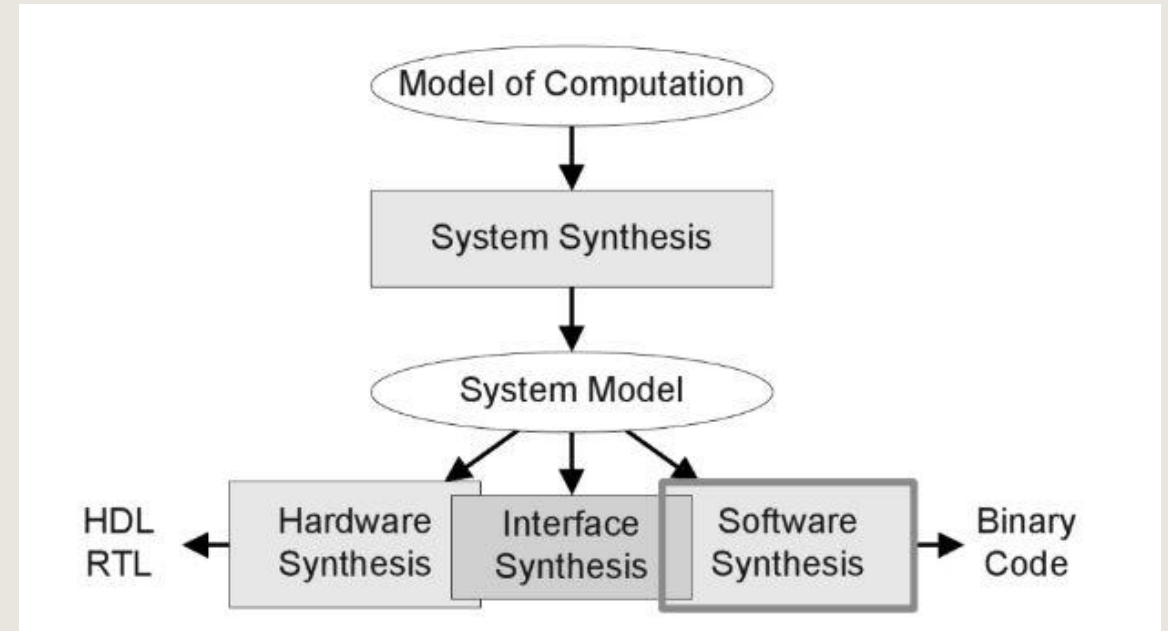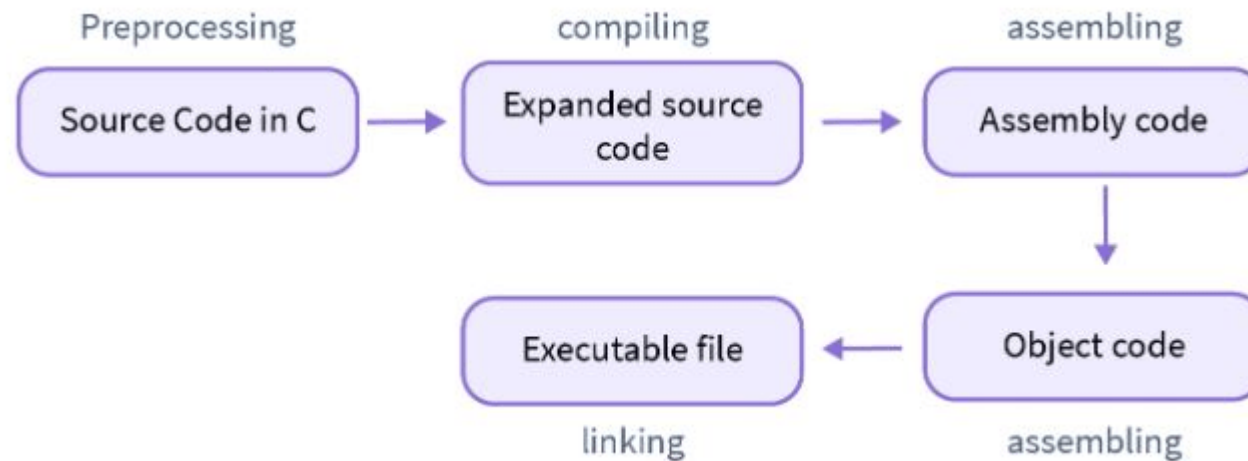
# APPROACHES TO SOFTWARE SYNTHESIS

- Embedded systems : part of the physical control process.
- Embedded software can implement a control loop of the physical process through getting readings from the sensors.
- Therefore, the software is specific to the underlying hardware, in order to achieve that embedded systems use specialized components (hardware accelerators, processors, DPSs) with communication schemes.

# PROGRAMMING IN HIGH LEVEL-LANGUAGES

- What is High level Languages:
- different from human languages
- How to write a program
- computers can only understand binary numbers.
- translators to lower the level of programming language

# COMPILERS

- The high-level languages must be translated into low-level
- The compiler create an executable binary directly
- Assembly code .S
- Linker operations
- Libraries

| Preprocessing | compiling | assembling |
|---|---|---|
| Source Code in C → | Expanded source code → | Assembly code |

| linking | | assembling |
|---|---|---|
| Executable file ← | | Object code |

[2]

# PRE-PROCESSING

- Pre-processing is the first step in the compilation process in C
- It is performed by the pre-processor tool
- All the statements with the symbol of #

- Steps,
1. Comments removal :

```
 1  /************************************************************************
 2
 3  |              |              |      Online C Compiler.
 4  |              |      Code, Compile, Run and Debug C program online.
 5  Write your code in this editor and press "Run" button to compile and execute it.
 6
 7  ************************************************************************/
 8
 9  #include <stdio.h>
10  #define GRAV 9.8
11  int main()
12  {
13      int a=4;   //accleraration
14      int v=43; //velocity
15      printf("Value of the Gravity is: %f ",GRAV);
16
17      if (v>a){
18          printf("\n%d",v);
19      }
20
21      return 0;
22  }
23
```
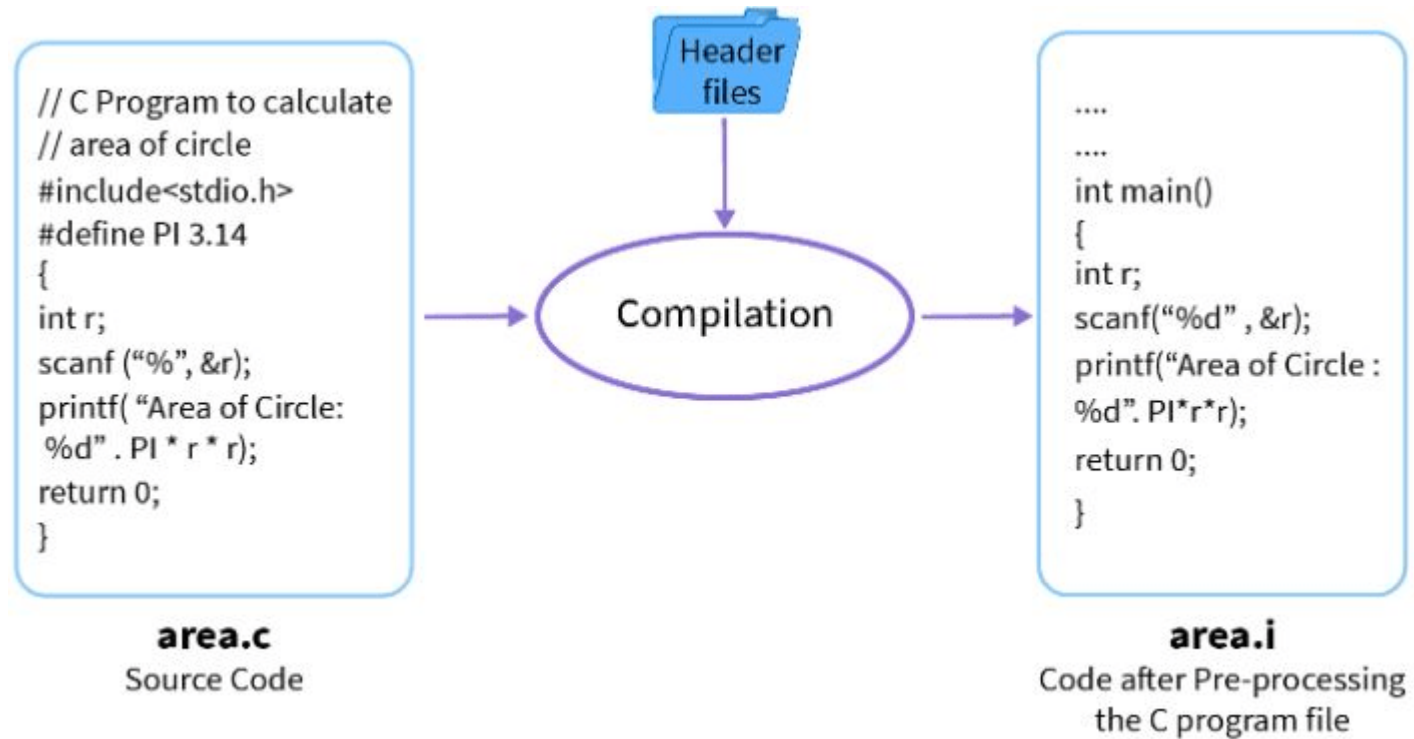
# 2. Macros expansion:

```
1
2
3    #include <stdio.h>
4    #define GRAV 9.8
5    int main()
6 ~  {
7        int a=4;
8        int v=43; |
9        printf("Value of the Gravity is: %f ",GRAV);
10
11 ~     if (v>a){
12           printf("\n%d",v);
13       }
14
15       return 0;
16   }
17
```

## 3. File inclusion:
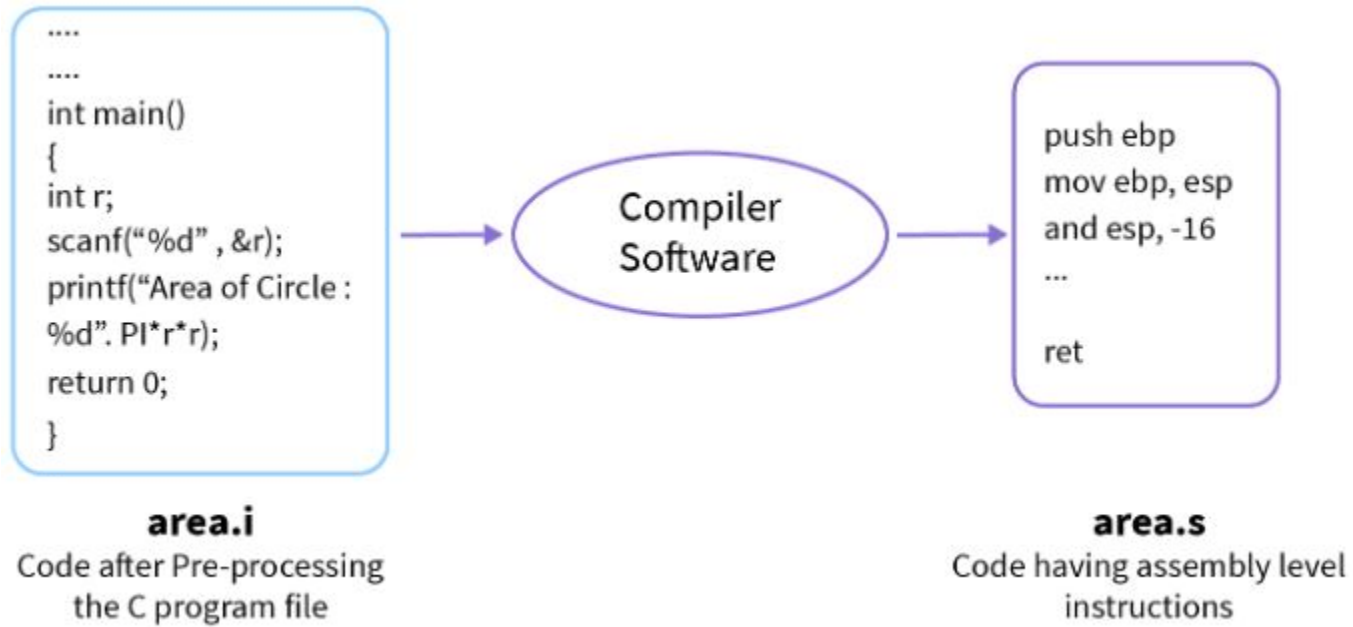
```c
1
2
3  #include <stdio.h>
4  int main()
5  {
6      int a=4;
7      int v=43;
8      int GRAV= 9.8;
9      printf("Value of the Gravity is: %f ",GRAV);
10
11     if (v>a){
12         printf("\n%d",v);
13     }
14
15     return 0;
16 }
17
```

# 3.Conditional compilation:



```
// C Program to calculate
// area of circle
#include<stdio.h>
#define PI 3.14
{
int r;
scanf ("%", &r);
printf( "Area of Circle:
 %d" . PI * r * r);
return 0;
}
```

**area.c**
Source Code

Header files

Compilation

```
....
....
int main()
{
int r;
scanf("%d" , &r);
printf("Area of Circle :
%d". PI*r*r);
return 0;
}
```

**area.i**
Code after Pre-processing
the C program file

[2]

# 5.Compiling:



```
....
....
int main()
{
int r;
scanf("%d" , &r);
printf("Area of Circle :
%d". PI*r*r);
return 0;

}
```

**area.i**
Code after Pre-processing
the C program file

Compiler Software

```
push ebp
mov ebp, esp
and esp, -16
...

ret
```

**area.s**
Code having assembly level
instructions

[2]

# 6.Assembling:



```
push ebp
mov ebp, esp
and esp, -16
...

ret
```

**area.s**
Code having assembly level
instructions

Assembler

```
0101010101
0100000101
0101110100
1010011000
1010010010
```

**area.o**
Object file haveing binary code
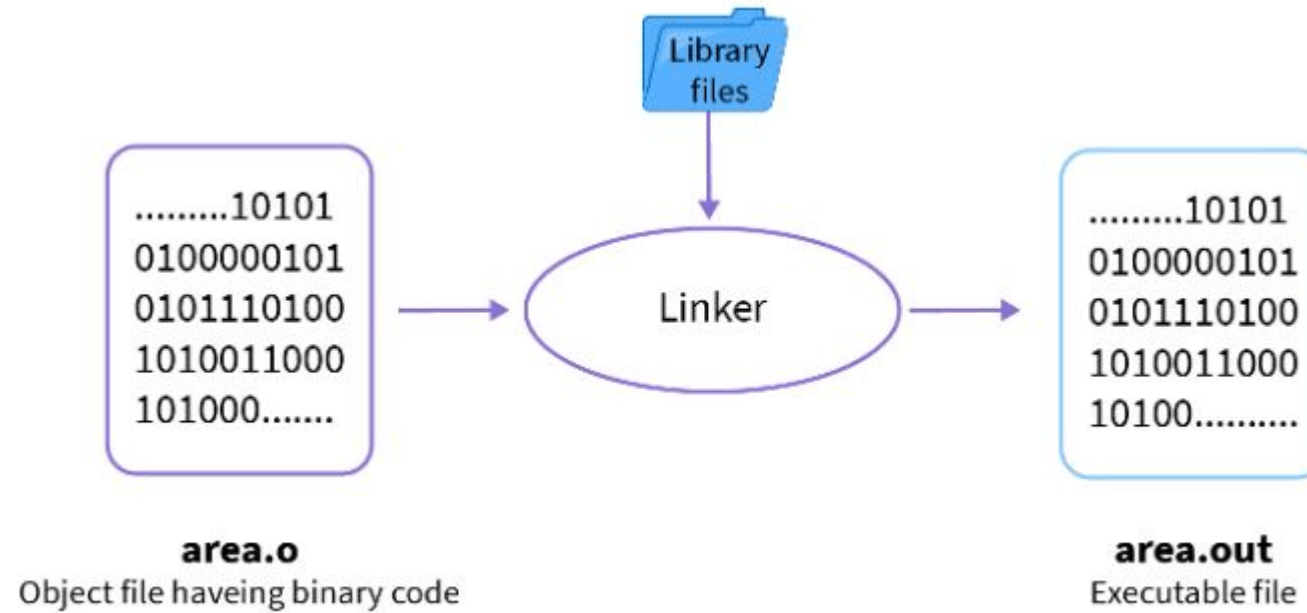
[2]

# 7.Linking



area.o
Object file haveing binary code
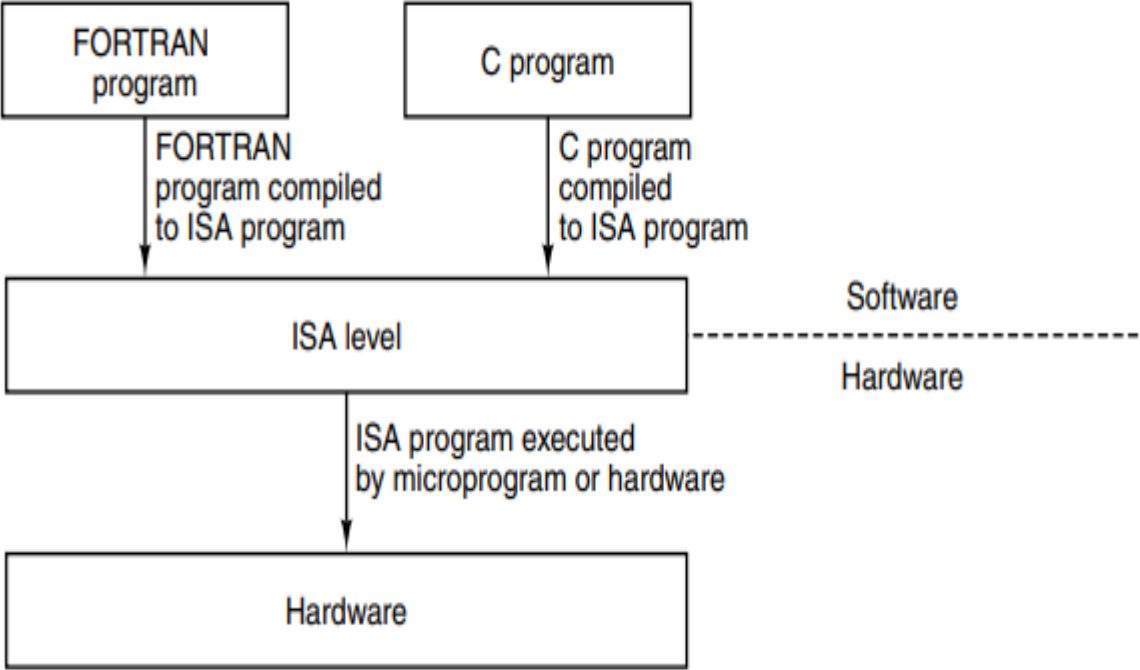
area.out
Executable file

[2]

# INSTRUCTION SET ARCHITECTURE (ISA)

- produce a way to connect software to hardware.
-  defines the type of instructions to be followed by the processor

- Type of operations

1. Arithmetic /logic instructions:

2. Data Transfer instructions:

3. Branch and jump instructions:

Software Synthesis for embedded processors

# ISA

# REDUCED INSTRUCTION SET COMPUTER (RISC)

- At the start of RISC, the people who were working on microprogramming tools began to rethink the architectural design principles trying to close the "semantic gap" in high-level programming languages and microinstructions.

- optimizing compilers that can be used to compile high-level languages into instructions.

- These instructions are simpler

- Key principles of RISC

1) Functions should be simple.

2) Microinstructions should not be faster than simple instructions,

3) Moving software into microcode does not improve its performance or functionality

4) The RISC architecture prioritizes simple instruction decoding and pipeline execution Use of compiler technology to simplify instructions.

# ARM CPU ARCHITECTURE

- ARM is a RISC architecture.
- The ARM ISA.
- All ARM instructions are 32-bit long
- three-operand encoding.
- large register file with 16 general-purpose registers, allowing pipelining

**A. Registers:**
- 16 general-purpose registers in the user mode.
- Register 15
- Register 14 .
- Register 13

B.    **ARM architecture exceptions**

# CONCLUSION

# THANK YOU

Lochana Abhayawardana

Lochana.abhayawardana@stud.hshl.de