SISTEMI EMBEDDED
AA 2013/2014

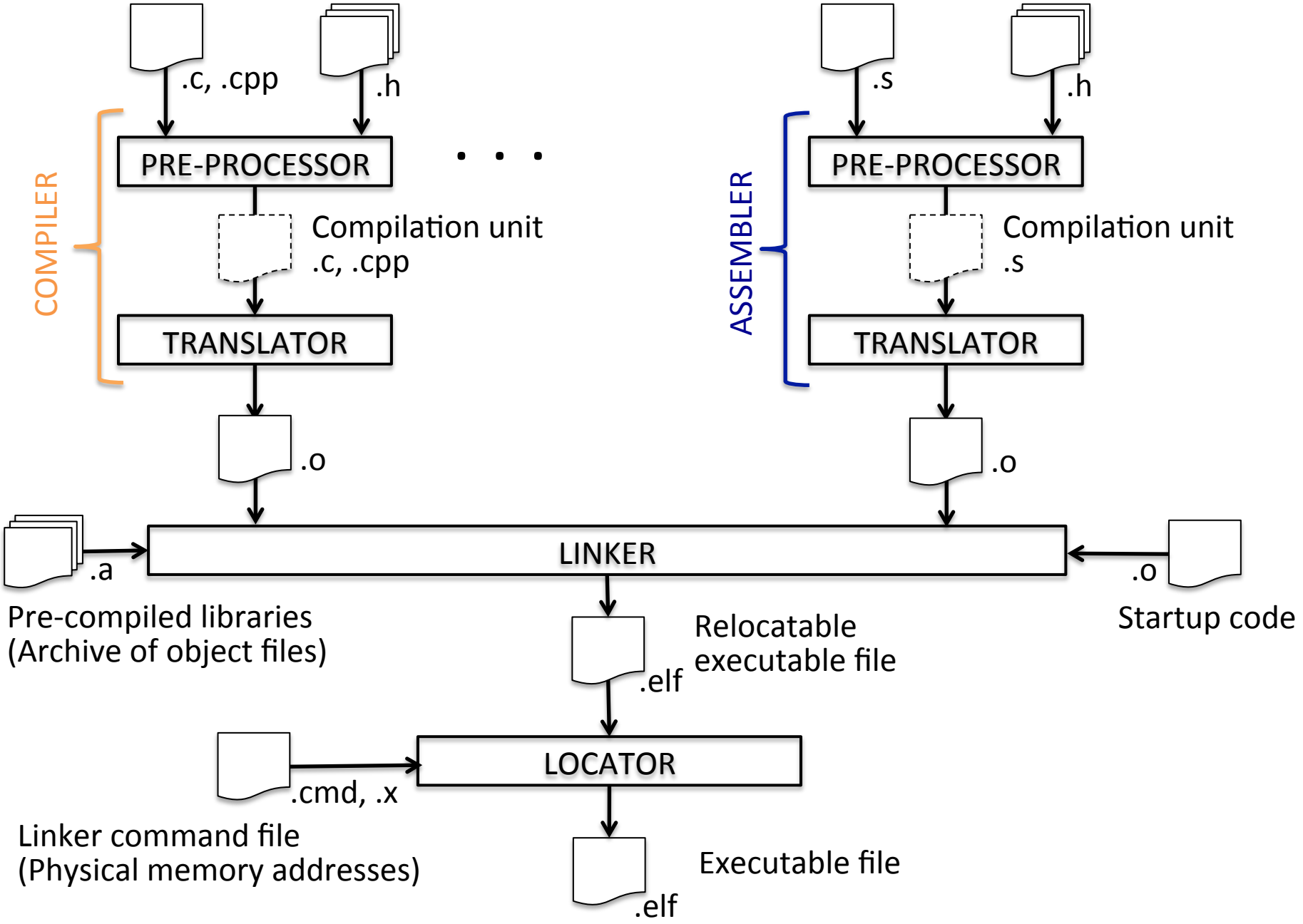
Nios II processor

Building process

Code optimisation

Federico Baronti

The Build Process



Compiler (1)

- Compilation unit: a single text file obtained from the related source file after being processed (file inclusion, string manipulation, …) by the pre-processor
- A Compiler translates a high-level language as C into a list of OPCODES (instructions) **specific** for the used processor
  - In embedded system programming, we use a cross-compiler, as the tool runs on a different processor architecture than the generated code

Compiler (2)

- The result of the compilation is an **object** file (binary file)
- It is organized in sections. The basic ones are:
  - *text* (containing the code)
  - *data* (contained initialized static/global variables)
  - *bss* (contained non-initialized static/global variables, which will be initialized to zero by the startup code)

Compiler (3)

- An object file also contains a header that describes the following sections, and some tables.
  - The table of symbols contains the names and locations (section and offset) of all the functions and variables referenced in the source file.
  - Rows of this table may be incomplete, because some functions and/or variables can be defined in a different file
  - It is up to linker to complete the missing information

Linker and Loader (1)

- Combine all the object file generating a new object file where the table of symbols does not contain unresolved names (otherwise an error is generated; an error occurs also if a symbol is defined in more than one object file)
- The generated object file is a **relocatable** copy of the program. All the sections starts from address zero at this point

Linker and Loader (2)

- A physical memory address must be assigned to the start address of each section
- This mapping is determined by a *linker script* (or command file)
  - It contains for each section the corresponding starting physical memory address

Startup code

- Small portion of assembly code (crt0.s) which prepares the processor for executing the remainder part of the program. It basically does the following:
  1. Disables all the interrupts
  2. Initializes all the initialized variables with their initialized values (stored in a ROM memory)
  3. Initializes with zero all the variables in the bss section
  4. Initializes the stack pointer
  5. Call *main*
  6. After "Call *main*" is present an infinite loop, in case the *main* function will return

Make and makefile (1)

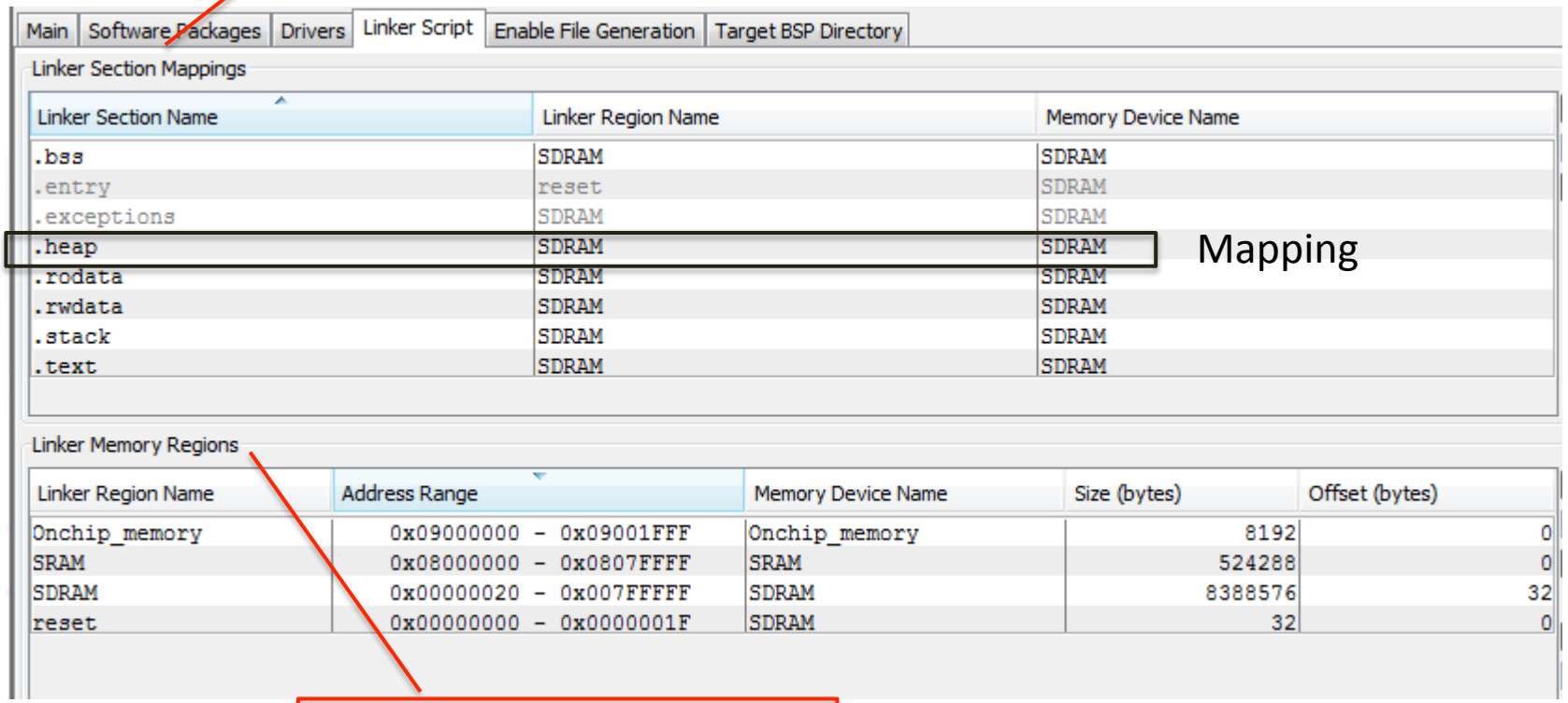
- The build process requires executing the compile command for all the source file and then invoking the linker tool
- This process can be automatized using the make tool. Its input is the makefile file, which is a list of rules:
  - target: prerequisite
    command

Make and makefile (2)

- The build process requires executing the compile command for all the source file and then invoking the linker tool
- This process can be automatized using the **make** tool. Its input is the makefile file, which is a list of rules:
  - *target*: *prerequisite*
    command
  - The *target* is what is going to be produced by the rule. The *prerequisite* are the files that must exist before the target can be created using the shell *command*
  - A target is recreated only if the prerequisite files are more recent than the target file
- The makefile can be autogenerated by the Integrated Devoloping Environment

Memory mapping

Corresponds to virtual memories where the linker place code, data, stack, heap,...



Corresponds to physical memories created with Qsys.

Automatic code placement

- The reset handler code is always placed at the base of the *.reset* partition. The general exception funnel code is always the first code in the section that contains the exception address. By default, the remaining code and data are divided into the following output sections:
  - .text All remaining code
  - .rodata The read-only data
  - .rwdata Read-write data
  - .bss Zero-initialized data

Manually-controlled placement

- In your program source code, you can specify a target memory section for each piece of code. In C or C++, you can use the section attribute. This attribute must be placed in a function prototype; you cannot place it in the function declaration itself

```
/* data should be initialized when using the section attribute */
int foo __attribute__ ((section (”my_section"))) = 0;
void bar (void) __attribute__ ((section (".sdram.txt")));
void bar (void)
{
foo++;
}
```

Stack and heap placement

- By default, the heap and stack are placed in the same memory partition as the .rwdata section
- The stack grows downwards (toward lower addresses) from the end of the section
- The heap grows upwards from the last used memory in the .rwdata section
- You can control the placement of the heap and stack by manipulating BSP settings
- By default, the HAL performs no stack or heap checking. This makes function calls and memory allocation faster, but it means that malloc() (in C) and new (in C++) are unable to detect heap exhaustion
- You can enable run-time stack checking by manipulating BSP settings. With stack checking on, malloc() and new() can detect heap exhaustion
- Stack checking has performance costs. If you choose to leave stack checking turned off, you must code your program so as to ensure that it operates within the limits of available heap and stack memory

Controlling code size (1)

- Very important to reduce memory costs
- The HAL environment includes only the features used by the application
  - If the Nios II hardware system contains exactly the peripherals used by the application, the HAL contains only the drivers necessary to control the hardware

Controlling code size (2)

- Available options to reduce code footprint (size)
  - Compiler optmisation
    - Some optimisation flags which control the trade-off between increasing speed and reducing memory use
  - Reduced device driver
    - Lighter device driver version (slower and less functions)

Peripheral	Small Footprint Behavior
UART	Polled operation, rather than IRQ-driven
JTAG UART	Polled operation, rather than IRQ-driven
Common flash interface controller	Driver excluded in small footprint mode
LCD module controller	Driver excluded in small footprint mode
EPCS serial configuration device	Driver excluded in small footprint mode

Controlling code size (3)

- Available options to reduce code footprint (size)
  - Reduce the File Descriptor Pool
    - The file descriptors that access character mode devices and files are allocated from a file descriptor pool. It can be changed through a BSP setting. The default is 32
  - Use /dev/null
    - At boot time, standard input, standard output, and standard error are all directed towards the null device, that is, /dev/null. After all drivers are installed, these streams are redirected to the channels configured in the HAL
    - The footprint of the code that performs this redirection is small, but you can eliminate it entirely by selecting null for stdin, stdout, and stderr when stdio is not used
    - You can control the assignment of stdin, stdout, and stderr channels by manipulating BSP settings

Controlling code size (4)

- Available options to reduce code footprint (size)
  - Use the Small newlib C Library. Some limitations:
    - No floating-point support for printf() family of routines
    - No support for scanf() family of routines
    - No support for seeking
    - No support for opening/closing FILE *. Only pre-opened stdout, stderr, and stdin are available
    - No buffering of stdio.h output routines
    - No stdio.h input routines
    - …
  - Use UNIX-Style File I/O fully omitting the C library
    - Standard I/O C functions can be emulated by application code

Controlling code size (5)

- Available options to reduce code footprint (size)
  - Use the Minimal Character-Mode API
    - If you can limit your use of character-mode I/O to very simple features, you can reduce code footprint by using the minimal character-mode API
    - This API includes the following functions:
      - alt_printf()
      - alt_putchar()
      - alt_putstr()
      - alt_getchar()
    - These functions are appropriate if the program only needs to accept command strings and send simple text messages.