SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

ENSC254 – Assembly Language

June 2020

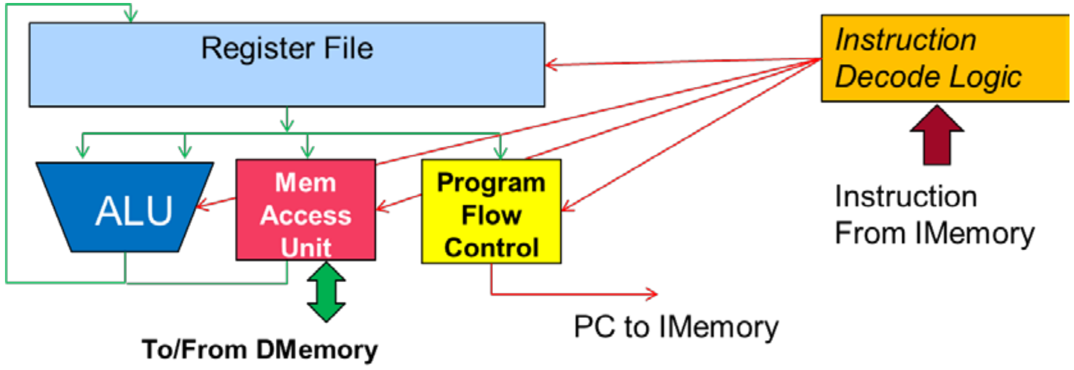
School of Engineering Science
Simon Fraser University
Burnaby, BC, Canada

Brief Review

Concepts you need from previous lectures:

1) What is a RISC Machine? What is meant as Load/Store ? What is a Harvard Architecture?
2) What is a Register File? What is an ALU? What is a PC (Program Counter)? What is a LR (Link Register)

3) How to encode Instructions? What are RRR, RRI formats?
4) What is the effect of the following kinds of instructions on the picture below:
   a. Data Processing Instructions
   b. Load/Store Instructions
   c. Branch instructions



ASSEMBLY Language

- In order to make it possible for us to write code with some level of efficiency, we need to abstract from the machine instruction format outlined before, and use formats that are friendlier for our minds, such as those with *string mnemonics*.
- That is why we associate readable names to all the fields that are just bit collections to the processor
- *ASSEMBLY language, is a human-readable rendition of a machine language*. Although strictly similar (especially in RISC machines), ***assembly is not the same as machine code***, and a single ASSEMBLY pseudo-instruction can be expanded into multiple machine instructions.
    - Branch instructions in Assembly usually refer to *labels* in the code and not numeric addresses, because such addresses are often not considered at the time of writing, and in any case the machine code for a branch instruction contains calculated offset information, not information for an actual address.
    - Labels in Assembly are symbolic and not yet resolved to memory addresses
    - Moreover, assembly language also utilize DIRECTIVES, sometimes to guide the generation of machine code

Example:

ARM Source Assembly CODE

```
    mov r3, #0
    mov r1, #7                ; Operand 1
    mov r2, #3                ; Operand 2  r2 will be our loop index and mult operand
loop:
    subs r2,r2,#1             ; decrement multiplier operand
    add r3,r3,r1
    bne loop                  ; iterate until r2=0.
stop:
    b stop
```

ARM DISASSEMBLED CODE

```
0x00000000  E3A03000  MOV     R3,#0x00000000
0x00000004  E3A01007  MOV     R1,#0x00000007
0x00000008  E3A02003  MOV     R2,#0x00000003
0x0000000C  E2522001  SUBS    R2,R2,#0x00000001
0x00000010  E0833001  ADD     R3,R3,R1
0x00000014  1AFFFFFC  BNE     #-8
0x00000018  EAFFFFFE  B       #0
```

Please note how labels represent addresses (i.e. memory locations)

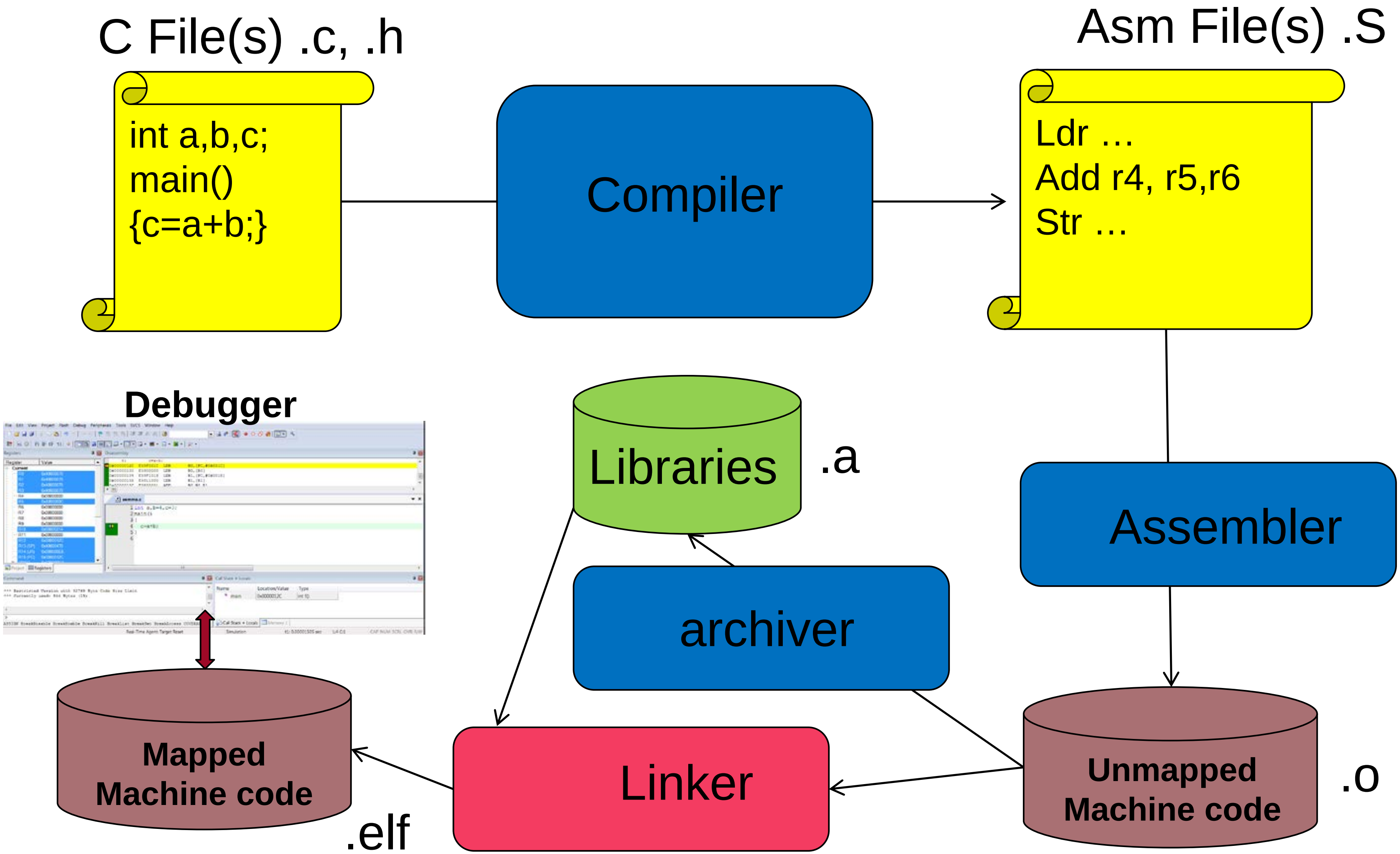
High level Programming Languages

- Of course, using Assembly to describe complex computation procedures is not efficient, and we would waste an immense amount of programmer time to write a simple program, which would be uneconomical

- For this reason, we write down high-level software in high-level languages (such as C or C++, or Java which is a bit different)
  - Humans can be way more efficient in writing C/C++ than they can be in writing Assembly Language
  - We can write software to automatically translate C/C++ into Assembly and then machine code. Software historically hasn't been as good as a human in optimizing every single line, but we are happy to pay the price in order to write at a higher level
  - MOST IMPORTANTLY, C and C++ are independent from an ISA, while Assembly language is strongly dependent on an ISA

As an outcome of this course, you should be able to understand and predict how C is turned into Assembly, and be able to understand the optimization tradeoffs in this translation

Compilation Flow



Cross-Compilation

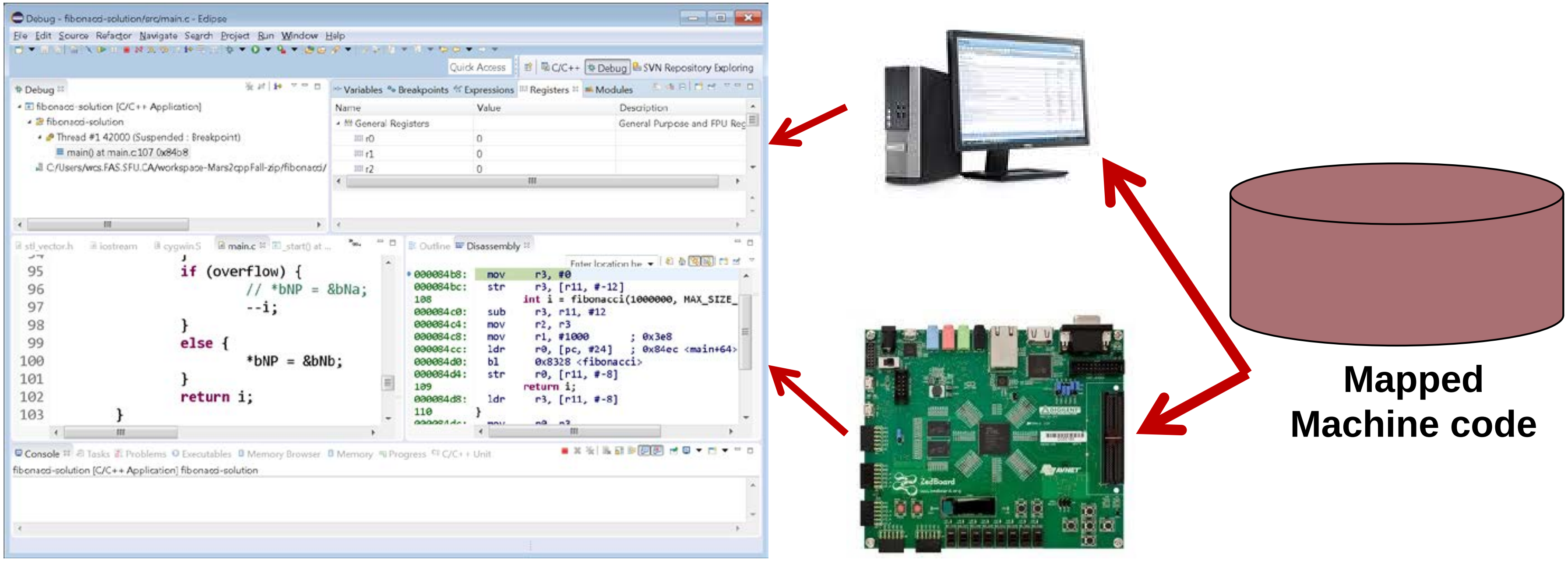
- The previous slide also essentially describes the «classic» compilation flow, by which executables are built in any Operating System and environment (Notably, the GNU Compilation tool-suite environment)

- *Our case is a bit different, in that we are using a host desktop machine to compile code for an embedded processor. That is a processor in a specific* ***Integrated circuit*** *on a standalone board, not part of our desktop system*

- ***We define CROSS-COMPILER a software tool that produces code not for the processor on which it is running but for a different, external, usually embedded, processor.***

- While the first steps of the flow are identical, the debugging is different: we cannot run the ARM 32-bit machine code natively on our Intel x86 processor!!!!! (Or not even on the ARM Cortex-M embedded in some of our electronics for that matter)

Cross-Compilers, ISS, JTAG



- When cross-compiling to an embedded processor, we are left with two debug options:
  1. Write simulation software, capable of understanding and emulating the embedded processor core on the host desktop system. This emulator is called an ***ISS (Instruction Set Simulator). The KEIL uVision toolsuite and a gdb that I built include simulators!***
  2. Set up some connection (USB, Ethernet …) to the IC on its board, and copy to the desktop the program status read from the board. Such practice uses a JTAG-based hardware debugger. ***The Xilinx SDK supports JTAG (as does KEIL toolsuite)!***

Why Assembly ?

- At times, it is important to control specific features of a given hardware system
    - Example 1, to write specific control software (DRIVER) for a given peripheral
    - Example 2, to use some specific function unit of a processor that a compiler is not able to generate assembly for from the C code
    - Example 3, to perform specific optimizations on a very critical piece of code

- In fact, C can be used at such a low level, that most drivers could be written in C. Still, assembly can allow full exposure to the CPU and of the system features, and for students that is a very useful learning opportunity
- You don't need to think of it as a ***PROGRAMMING LANGUAGE***, but can instead think of it as a way to *see and debug what is actually happening on your processor*