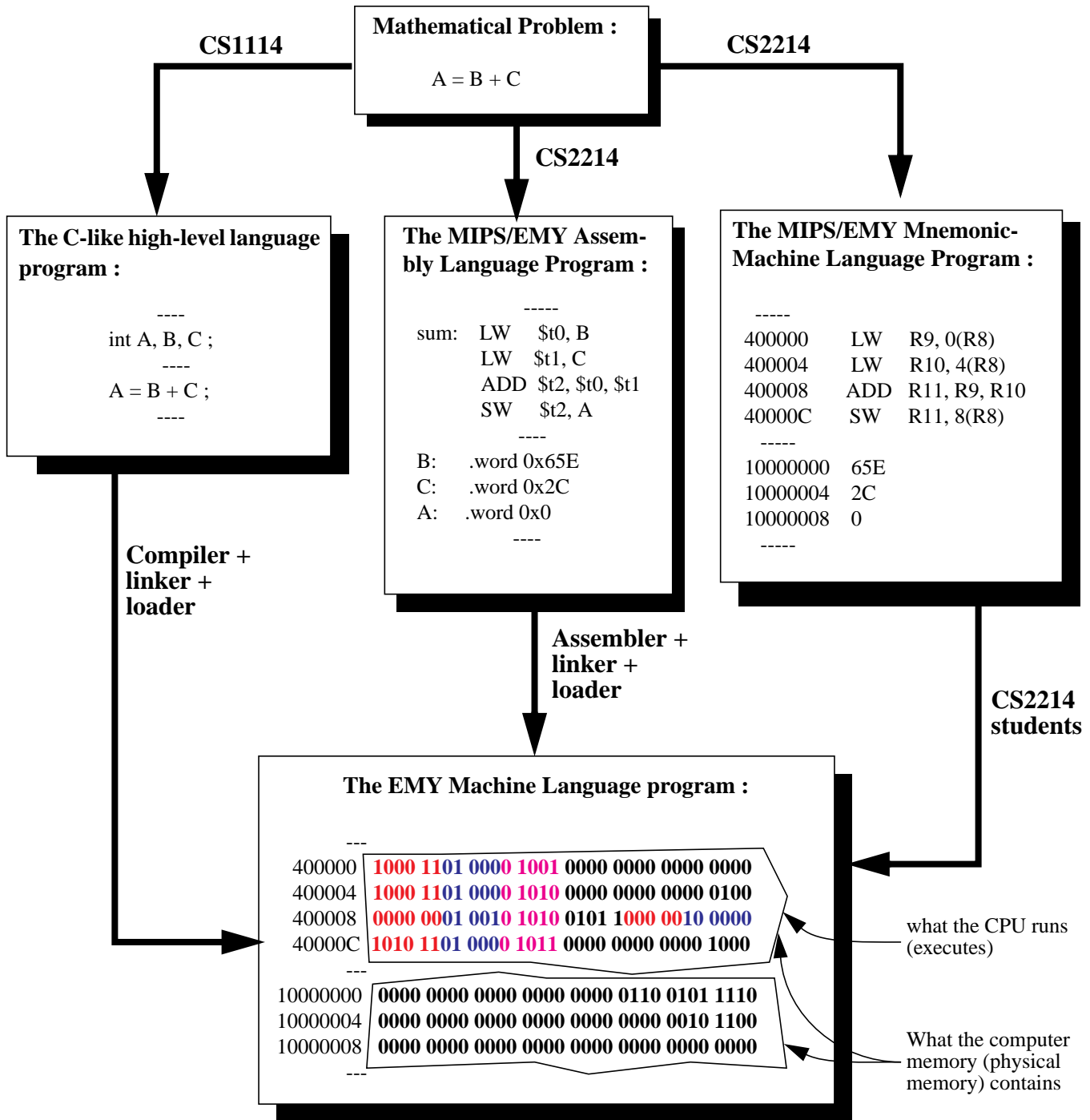


EMY MNEMONIC MACHINE LANGUAGE PROGRAMMING EXAMPLES

Programming at different levels



1) Add two numbers which are in memory locations 10000000 and 10000004 and then place the result in location 10000008. This piece of mnemonic machine language program starts at location 400000 and implements the high-level language statement

$$A = B + C ;$$

```

400000 LW R9, 0(R8) # Load from memory to register : R9 <-- M[10000000], R9 <--- 65E
400004 LW R10, 4(R8) # Load from memory to register : R10 <-- M[10000004], R10 <--- 2C
400008 ADD R11, R9, R10 # Add two registers : R11 <-- R9 + R10, R11 <--- 65E + 2C
40000C SW R11, 8(R8) # Store from register to memory : M[10000008] <-- R11, M[10000008] <--- 68A
    ---      ---
1000000 65E # Variable B
1000004 2C # Variable C
1000008 0 # Variable A is stored after the code completes. Value 0 is before the execution
    ---      ---

```

NOTE 1 : The above piece of MIPS mnemonic machine language program

- ⇒ Has register R8 already initialized to 10000000
- ⇒ Loads the content of memory location 10000000 to register R9,
- ⇒ Loads the content of memory location 10000004 to register R10,
- ⇒ Adds the contents of register R9 and register R10 and places the result in R11,
- ⇒ Stores the result, 68A, in R11 to memory location 10000008

NOTE 2 : A high-level (HL) statement specifies an operation (an addition) on variables (A, B, C). A variable in a HL statement corresponds to a memory location in the machine language program. A HL statement is implemented by machine language instructions. A machine language instruction specifies one or more architectural operations. Thus, a HL operation is implemented by a number of architectural operations.

NOTE 3 : The memory is passive, i.e. it cannot manipulate its locations. In other words, the memory cannot perform architectural operations on data. It can only keep instructions and data. Thus, another unit is implied in machine language programs to manipulate memory locations. That implied unit becomes visible in microarchitecture : the CPU. It is the CPU that executes instruction. It is the CPU that performs architectural operations.

Clearly, the CPU has to receive (read) an instruction from the memory first, to determine which architectural operation to perform. Reading the memory to bring an instruction to the CPU is called instruction fetch. Data also reside in the memory. Then, there must be data transfers between the memory and the CPU before an operation (a data read) and maybe another data transfer from the CPU to the memory after an operation (a data write). The CPU has registers to keep data before and after an operation.

Some computers require explicit instructions to read data to the CPU (to a register) from the memory : LOAD instructions (LW for the EMY) and explicit instructions to write data (from a register) to the memory : STORE instructions (SW for the EMY). Between the LOADs and the STORE, instructions are used to perform real operations of the HL program, for example the ADD in the above piece of program. Now, we see why we need four instructions to implement the “A= B + C” statement above. Note that the ADD specifies only registers (not memory locations).

NOTE 4 : RISC computers require explicit instructions to read data to the CPU from the memory : LOAD instructions (LW for EMY) and explicit instructions to write data to the memory : STORE instructions (SW for EMY). Only LOADs and STOREs can access the memory for data. This kind of architectures is called LOAD/STORE architectures. Their arithmetic/logic/floating-point instructions (ADD, OR, ADD.S) can specify only registers (not memory locations). This is a distinctive feature of RISC machines.

CISC computers have more complex arithmetic/logic/floating-point instructions which can specify memory loca-

tions, rather than registers. In some CISC architectures, the “A = B + C” statement can be implemented by **one** instruction (not four). The single instruction accesses the memory for the data reads and writes, without needing any LW or SW.

Thus, one of the fundamental differences between RISC and CISC systems is that RISC machine language programs tend to have more instructions than CISC machine language programs do.

NOTE 5 : A convention used in machine language programming is that data elements are grouped together for efficient memory management. We place the data group separate from the program as done in the above piece of program.

NOTE 6 : As we know application programs and their data are stored in the user space, not in the kernel (system) space. This a common practice now and the EMY architecture requires that user programs and user data are stored in the user space of the memory : 00000000 - 7FFFFFFF. The EMY enforces it in hardware. In addition, there is a software convention (not enforced by the hardware) that user programs are stored starting at 400000 and user data are stored starting at 10000000 in the memory. The piece of program on the first page follows this software convention.

NOTE 7 : Another EMY software convention is the usage of registers, i.e. which register can be used at which point in a mnemonic machine language program. We already know that registers R8 through R25 are used for routine (temporary) calculations. A more detailed register usage convention description is as follows :

- R1 is reserved for the assembler
- R2 and R3 are used to return results from functions
- R4-R7 are used to pass parameters to functions
- R26 and R27 are reserved for the OS
- R29 is the stack pointer, pointing at full top. The stack grows towards lower addresses.
- R28 and R30 are special pointers: Stack grows from higher addresses to lower addresses.

- R8-R15 and R24-R25 contain values which will have a short life and caller saves them if necessary
- R16-R23 contain values which will have a longer life and the callee saves them
- **Again, we will use R8 to R25 during routine calculations.**

See also page 88 of the textbook and the “Green Card” for the software renaming of MIPS registers for assembly language programming. The piece of program on the first page also follows these.

NOTE 8 : The table below shows the values of registers and memory locations used and the type of memory accesses made by the program on the first page. Note that the values are recorded after an instruction is completely executed :

Instruction	PC	R8	R9	R10	R11	M[10000000]	M[10000004]	M[10000008]	Memory Accesses
Initial	400000	10000000	?	?	?	65E	2C	0	---
LW R9, 0(R8)	400004	NS	65E	NS	NS	NS	NS	NS	Ins read and data read
LW R10, 4(R8)	400008	NS	NS	2C	NS	NS	NS	NS	Ins read and data read
ADD R11, R9, R10	40000C	NS	NS	NS	68A	NS	NS	NS	Ins read
SW R11, 8(R8)	400010	NS	NS	NS	NS	NS	NS	68A	Ins read and data write

When this piece of program is executed, seven (7) memory accesses are made by the CPU, four for reading (fetching) instructions and three for data. Of these three memory accesses for data, two are to read data and one is to write data.

2) Write a mnemonic machine language program that implements the following high-level language statement :

$$A = B | C ;$$

The statement ORs two variables and stores the result in another variable. Assume that variable A is in memory location 10000004, “B” is in R8 and “C” is in memory location 10000000, containing 4A0. Assume also that R8 contains 27 and R9 contains 10000000 initially. This mnemonic machine language program starts at location 400000.

```

400000    LW    R10, 0(R9)    # Load from memory to register : R10 <-- M[10000000], R10 <--- 4A0
400004    OR    R11, R8, R10 # OR two registers : R11 <-- R8 | R10, R11 <--- 4A7
400008    SW    R11, 4(R9)   # Store from register to memory : M[10000004] <-- R11, M[10000004] <--- 4A7
---
10000000 4A0                # Variable C
100000004 0                 # Variable A is stored after the code completes. Value 0 is before the execution

```

NOTE 1 : Following application and kernel storage requirements, the above subroutine which is an application subroutine is stored in the user space : 00000000 - 7FFFFFFF. In addition, following the software convention (not enforced by the hardware) the subroutine is stored starting at 400000.

NOTE 2 : The table below shows the values of registers and memory locations used and the type of memory accesses made by the program above. Note that the values are recorded after an instruction is completely executed :

Instruction	PC	R8	R9	R10	R11	M[10000000]	M[10000004]	Memory Accesses
Initial	400000	27	10000000	?	?	4A0	0	---
LW R10, 0(R9)	400004	NS	NS	4A0	NS	NS	NS	Ins read and data read
OR R11, R8, R10	400008	NS	NS	NS	4A7	NS	NS	Ins read
SW R11, 4(R9)	40000C	NS	NS	NS	NS	NS	4A7	Ins read and data write

3) Write a mnemonic machine language **subroutine** that takes the absolute value of number “k” passed to the subroutine in R4. The result is returned in R2. The subroutine starts at 400400.

```

400400    ADD    R2, R4, R0    # The return register gets k, in case “k” is positive
400404    SLT    R8, R4, R0    # Is “k” less than 0, (is “k” negative) ?
400408    BEQ    R8, R0, 1     # If “k” is greater than or equal to 0, go to 400410 to return
40040C    SUB    R2, R0, R4    # “k” is less than 0. Take its 2’s complement
400410    JR     R31           # Return from the subroutine.

```

NOTE 1 : The **EMY architecture** uses integer numbers in the 2’s complement form and in the unsigned form. Given a number “k” in the 2’s complement form, “-k” is obtained by taking the 2’s complement of “k.” To calculate the 2’s complement of k, we subtract “k” from 0 as done above.

NOTE 2 : There are four important tasks that must be performed between a calling routine and a subroutine : a) passing parameters to the subroutine (R4 passes the parameter above), b) returning results from the subroutine (R2 returns the result from the subroutine), c) remembering where to return in the calling routine (R31 keeps the return address) and d) not destroying register values needed by the calling routine (the software conventions ensure that).

NOTE 3 : Assuming that the subroutine is passed $(-7)_{10}$ (R4 has $(-7)_{10}$), and the return point is 40020C (R31 has 40020C), the table below shows the values of registers and memory locations used and the type of memory accesses

made by the program above :

Instruction	PC	R2	R4	R8	R31	Memory Accesses
Initial	400400	?	$(-7)_{10}$?	400200C	---
ADD R2, R4, R0	400404	$(-7)_{10}$	NS	NS	NS	Instruction read
SLT R8, R4, R0	400408	NS	NS	1	NS	Instruction read
BEQ R8, R0, 1	40040C	NS	NS	NS	NS	Instruction read
SUB R2, R0, R4	400410	7	NS	NS	NS	Instruction read
JR R31	40020C	NS	NS	NS	NS	Instruction read

NOTE 4 : How would you change the above table if R4 had $(7)_{10}$?

NOTE 5 : Ignoring the “JR” instruction, obtain the corresponding C-like high-level language statement.

4) Write a subroutine for multiplying a number Y by a number Z both of which are always greater than 0. Y and Z are passed in registers R4 and R5, respectively. The subroutine starts at 401050. The result is returned in R2 to the calling program.

```

---      ---
401050  ADD  R8, R0, R0      # We clear R8
401054  ADD  R9, R4, R0      # We move R4, that is Y, to R9
401058  ADD  R8, R5, R8      # We add R5, Z, to R8 (add Z to temporary result)
40105C  ADDI R9, R9, (-1)10  # We subtract 1 from Y
401060  BNE  R9, R0, (-3)10  # Is it the end (is Y equal to zero) ? If not, go to 401058
401064  ADD  R2, R8, R0      # The end. We move the result to R2
401068  JR   R31            # We return from the subroutine
---      ---

```

NOTE 1 : Numbers Y and Z are multiplied by adding Z by Y times. In order to do Y successive additions, we use a **loop** which consists of three instructions in locations 401058, 40105C and 401060. In order to determine when to exit the loop, a counter is used (R9). When the counter reaches a certain number the loop is exited. Often, that number is 0 or the first negative number after the counter is 0 or the first positive number after the counter is 0. Above, the loop is exited when the counter (R9) is 0 and that is why the loop-branch-back instruction is “**BNE**”. It means if the counter is not equal to (zero), there is at least one more element and so branch back to the beginning of the loop.

NOTE 2 : What would we do if Y and Z were allowed to be positive and negative ?

5) A vector (a one dimensional array) with five elements is stored in locations 10002000 - 10002010. Write a subroutine to calculate the sum of the vector elements :

$$A = \sum_{i=1}^5 B(i)$$

The result is stored in 10002014 by the main program. The number of elements in the vector is in R4 initially and assume that it is always greater than 0. The starting address of the vector is in R5 initially. The piece of program starts at location 400600.

```

400600  ADD  R8, R0, R4      # "n" is moved to R8
400604  ADD  R9, R0, R0      # R9 is cleared
400608  ADD  R10, R0, R5     # R10 gets address of first element of the array
40060C  LW   R11, 0(R10)     # We read an element from the array pointed by R10
400610  ADD  R9, R9, R11     # Add this element to the previous sum
400614  ADDI R10, R10, 4     # We advance array pointer to the next element
400618  ADDI R8, R8, (-1)10  # We decrement the counter to check for end
40061C  BNE  R8, R0, (-5)10  # If not the end, we go back to location 40060C
400620  ADD  R2, R9, R0     # The end. Move the result to R2 to return it to main program
400624  JR   R31            # We return from the subroutine
---
10002000 A36           # First element of the vector
10002004 1B9           # Second element of the vector
10002008 8F3           # Third element of the vector
1000200C 6C           # Fourth element of the vector
10002010 28           # Fifth element of the vector
10002014 0            # The result is stored in this location
---

```

NOTE 1 : Working on arrays (vectors) also requires a loop to step through its elements. The program above uses a loop that spans instructions in locations 40060C through 40061C. The iterations of the loop are performed until the loop-ending counter, (R8) is 0.

NOTE 2 : There are mainly two addressing modes in machine languages to step through the elements of an array one by one : Indexed addressing (or its variations, such as the EMY 2-byte displacement) and memory indirect addressing. In the first mode, the index register (or the base register if it is the 2-byte displacement mode) is used to get the next element. Indexed addressing is a typical RISC addressing mode. In the memory indirect mode, a memory location, with the address of the next element is used to get the next element. This memory indirect addressing is a typical CISC addressing mode and not included in RISC architectures. Of course, a combination of these two addressing modes can also be used : the indirect indexed mode....

GENERAL NOTES : You may want to know **what** stores programs and data **where** in the memory.

In simple systems (such as embedded systems), it is the programmer who stores the programs and data in the memory and there is often no distinction between user programs and system programs. Therefore, there is one control mode. The hardware and the application suggest where the programmer should store them in the memory.

In more complex systems (such as PCs, servers, supercomputers), the programmer does NOT have any control. System programs, including the operating system (OS) are responsible for storing programs and data. System programs, hardware and the application altogether determine where to store them in the user space.

Important system programs making these decisions in large systems are the **linker** and **loader**. They interact with the operating systems, to store programs and data. Also, PCs and larger systems employ virtual memory, allowing users to have a small size physical memory backed up by a disk (a large virtual memory). The operating systems together with hardware modules called **memory management units** (MMUs) implement and support virtual memory.

Today's microprocessors contain two MMUs for high-speed memory access. One is used for instructions and one for data. An important component of the MMU is the translation-lookaside buffer (TLB). A microprocessor hence has two TLBs.

A TLB translates (converts) virtual addresses to physical addresses. TLBs and the conversion are discussed in the memory hierarchy chapter and can be ignored until that topic is covered.

For curious readers we note that all addresses we work on, for example 400000, 10000000, etc. are **not** real addresses, but virtual addresses. That is, all addresses the CPU generates are virtual addresses. They are translated to real, physical, addresses by the TLBs. Instruction virtual addresses are translated to instruction physical addresses by the instruction TLB and data virtual addresses are translated data physical addresses by the data TLB.