CC2511 Week 10

Upcoming schedule

- The schedule for the rest of the semester is:

Week number	Course content
10 (we are here)	Introduction to assembly language
11	Assembly part 2 (writing functions, interop with C)
12	Revision and exam preparation. The lab session is for assignment help.
13	No lecture. (Lecture time used for Q&A or assignment help.) The lab session is your assignment demonstrations.

Today: Assembly language

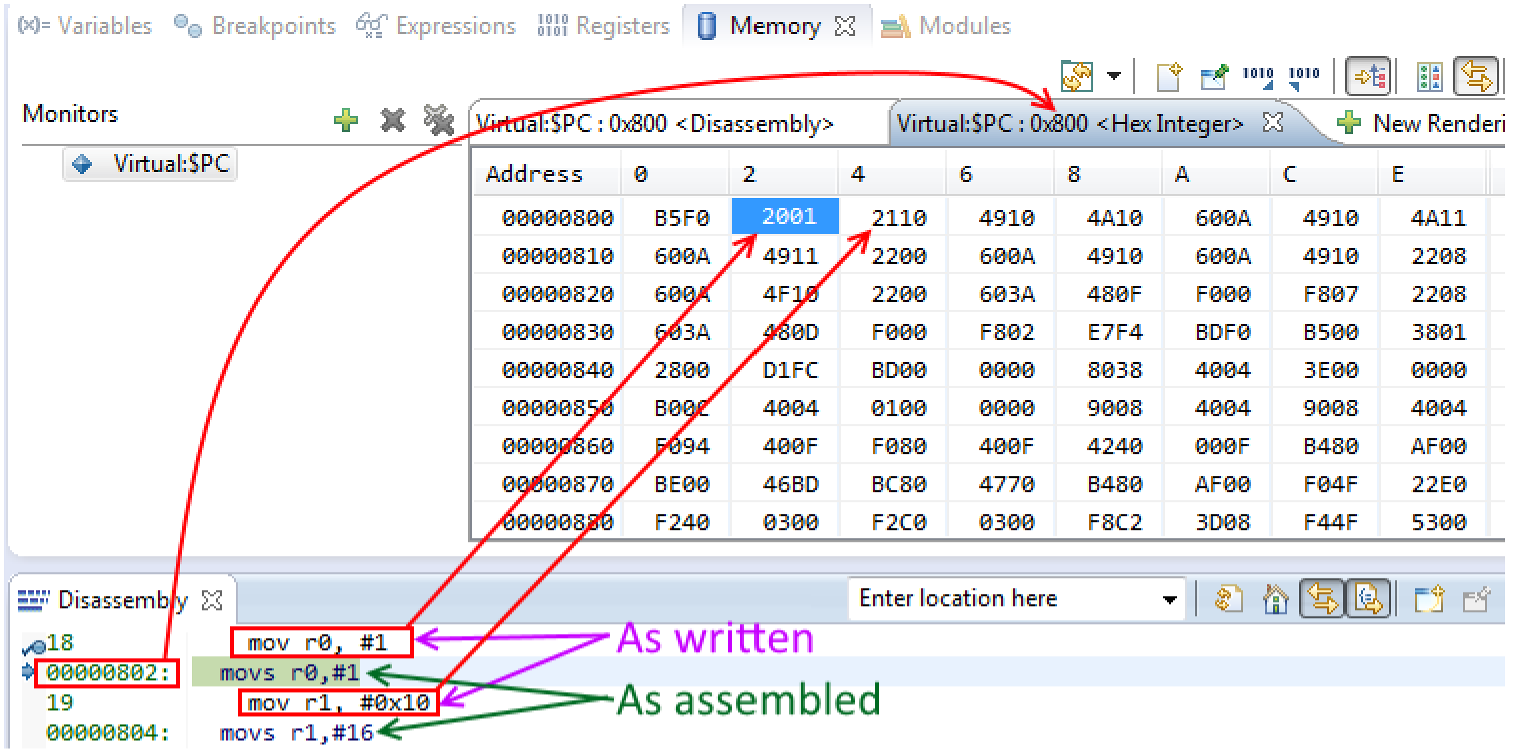
- What is assembly language?
- Why use it?
- Instructions: mov, ldr, str, add, sub, cmp, and conditional branch.

Assembly code

- Recall that the CPU contains an arithmetic-logic unit (ALU) which selects various logic gates according to which operation code (op-code) is loaded into its instruction register.
- The sequence of op-codes is called machine code.
- **Assembly code is a human-readable form of machine code.**
- There is an exact 1:1 mapping between assembly code and machine code.

Assembly code / machine code in memory

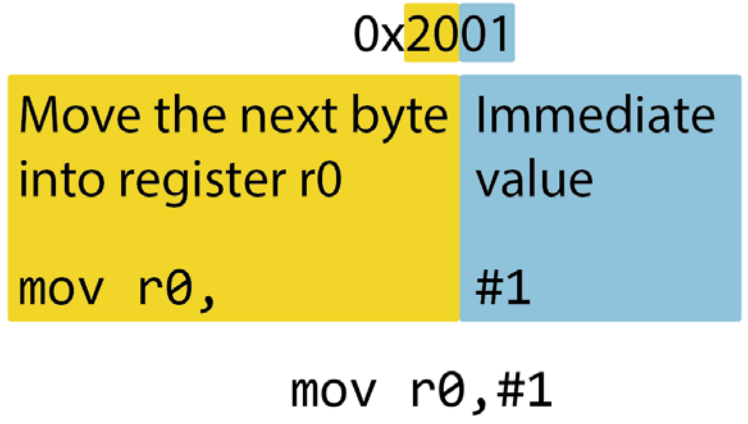
- Assembly code is a human readable form of machine code.



Address	0	2	4	6	8	A	C	E
00000800	B5F0	2001	2110	4910	4A10	600A	4910	4A11
00000810	600A	4911	2200	600A	4910	600A	4910	2208
00000820	600A	4F10	2200	603A	480F	F000	F807	2208
00000830	603A	480D	F000	F802	E7F4	BDF0	B500	3801
00000840	2800	D1FC	BD00	0000	8038	4004	3E00	0000
00000850	B00C	4004	0100	0000	9008	4004	9008	4004
00000860	F094	400F	F080	400F	4240	000F	B480	AF00
00000870	BE00	46BD	BC80	4770	B480	AF00	F04F	22E0
00000880	F240	0300	F2C0	0300	F8C2	3D08	F44F	5300

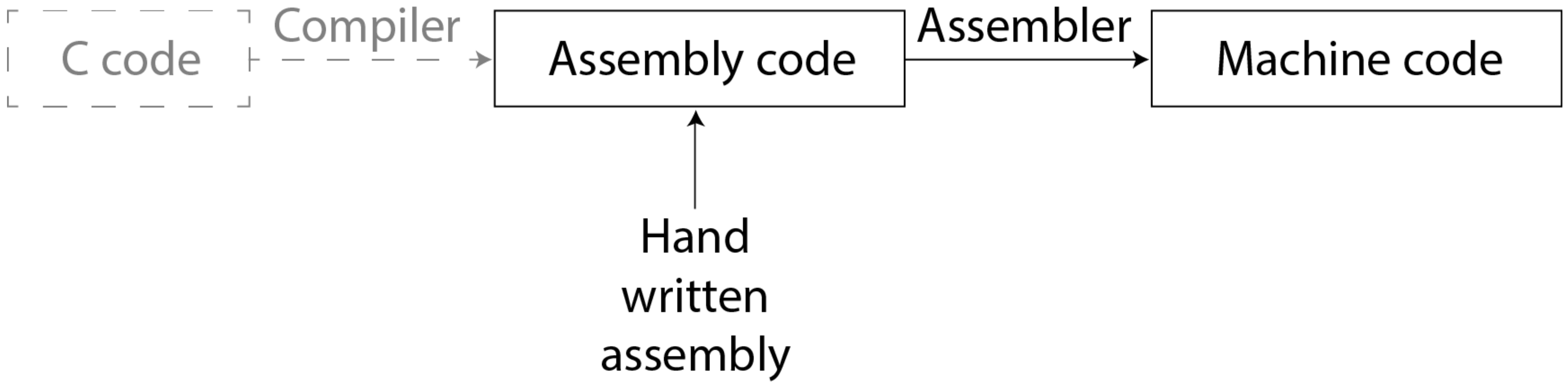
```
18          mov r0, #1
00000802:   movs r0,#1
19          mov r1, #0x10
00000804:   movs r1,#16
```

Example of decoding the opcodes:



The compilation flowchart

- Up until now, our assembly code has been machine-generated.
- Now, we'll write the assembly code directly.



Why write assembly code: Efficiency

- If the compiler can generate it, why bother writing it by hand?
- **Efficiency:** a skilled assembly programmer may be able to write more efficient code than a compiler.
- Often microprocessors (including ours) include specialised instructions for digital signal processing. A compiler might not be able to fully utilise these instructions.
  - On the other hand, compilers for popular architectures like ARM are very sophisticated and can in fact use many specialist instructions.

Why write assembly code: Flexibility

- **Flexibility**: there are assembly language constructs that cannot be expressed in C code.
- These are essential for low-level tasks such as configuring CPU modes and initialising hardware.
- Your compiler is probably not capable of using the full range of assembly instructions that your CPU supports.

Why learn assembly code: to have "mechanical sympathy"

- "Mechanical sympathy" is the idea that you will be a better programmer if you understand the low level details of the machine.
  - You should recognise when you are asking a lot vs when you are asking for a little.
- You cannot properly reason about performance or correctness unless you understand the low level details of the machine.

Why avoid assembly code: Programmer time

- A programmer will be much less productive in assembly code.
- There’s a lot of generic, repetitive code that the compiler would generate for you.
- The usual strategy: **write most of the program in C and only write the performance-sensitive or low-level parts in assembly.**
- We’ll learn how to mix C and assembly code, e.g. call assembly functions from C.

How to program in assembly

- **Each different CPU architecture will have a different assembly language.**
- To write in assembly, must have a detailed knowledge of the target CPU.

Our CPU architecture: ARM Cortex M4

- The basic architecture is called “ARMv7”.
  - This is the 7th revision of the ARM instruction set.
- ARMv7 is a 32-bit instruction set architecture. All ARMv7 op-codes are 32 bits long.
- The Cortex M4 is an ARMv7 customised for embedded systems.

ARM vs Thumb

- Using 32 bits for every instruction is wasteful of program memory.
- **Thumb** is a mixed 16 and 32 bit instruction set architecture designed to improve the CPU’s code density.
- ARM CPUs can be placed in “ARM mode” or “Thumb mode”.
- In Thumb mode, the shorter instructions are expanded into their ARM equivalents before being fed into the ALU.

Signalling Thumb mode

- **The least significant bit of the program counter signals whether the processor is in Thumb mode or ARM mode.**
- CPU instructions are “aligned” to 16-bit boundaries (i.e. all addresses are even). Thus the least significant bit is always zero and can be used as a flag.
- If the program counter (PC) has a 1 in the least significant bit:
  - The 1 is ignored, i.e. the address loaded is actually PC-1
  - The instruction at PC-1 is treated as Thumb instruction.

Cortex M4 and Thumb mode

- The Cortex M4 implements “Thumb2”, an enhanced version.
- **The Cortex M4 is always in Thumb mode** so every program address is always offset by 1.
- The Thumb2 instruction set is a subset of the ARM instruction set.
- Certain operations are not available in Thumb mode.
  - These operations are never available on the Cortex M4 (which is always in Thumb mode).

ARM is a load-store architecture

- **ARM is a load-store architecture.** This means arithmetic and logic instructions operate on registers only.

1. Read from memory into register(s).
2. Perform work, updating the register(s).
3. Write from the register(s) back into memory.

- There are no instructions that directly access memory (except load and store).

General purpose registers

- General purpose registers are used to hold variables. All arithmetic and logic instructions operate on these registers.
- **There are thirteen general purpose registers named R0, R1, … R12.**
- Each general purpose register is 32 bits long and must be accessed as a full 32 bit value.

Low registers and high registers

- Registers R0 to R7 are called “low registers” and have no restrictions on their use.
- Registers R8 to R12 are called “high registers” and cannot be accessed by most Thumb instructions.
  - They’re useful as fast temporary storage but must be swapped into a low register for actual work.

Summary of the architecture fundamentals

- CPU is always in Thumb mode (placing restrictions on which instructions can be used, and the program counter is offset by 1).
- Must load data into registers to perform work.
- Thirteen general purpose registers, of which only the first eight (R0 to R7) are available in all instructions.

Assembly syntax example

- Example: Load 6 into register r0

```
mov r0, #6
```

- The mov (“move”) instruction is the equivalent of a variable assignment: `r0 = 6;`
- Kinetis Design Studio uses “GNU syntax”: destination register first.
  - Other assemblers use different ordering.
  - To remember the ordering, imagine that the comma is an equals sign.

Immediate addressing

```
mov r0, #6
```

- The # at the front of the 6 indicates **immediate addressing**.
- **Immediate addressing means that the number should be read as a value** rather than an address.
- Can also use hexadecimal with a 0x prefix:

```
mov r1, #0x10
```

Immediate values must be <= 16 bits

- Thumb2 op-codes are a maximum of 32 bits long.
- The op-code must include:
  - The instruction
  - The registers involved
  - Any immediate values (constants)
- **There’s no way to fit a 32 bit immediate value and the instruction into a single op-code!**

Forms of the mov instruction

```
mov r0, #0x11       /* 8 bit immediate */
movw r0, #0x1111    /* 16 bit immediate */
mov r0, r1          /* set r0 to the contents of r1 */
```

- `mov` assembles into a 16 bit op-code (hence, 8 bit immediate).
- `movw` assembles into a 32 bit op-code (hence, 16 bit immediate).
- The w suffix means “word”, i.e. use word sized version of the instruction.

Loading 32-bit values: via two 16-bit moves

```
/* load 0x4004B00C into r1 */
movw r1, #0xB00C /* low bits first */
movt r1, #0x4004 /* move top (into the high bits) */
```

- The ordering must be `movw` then `movt`
- `movw` overwrites the high bits with zeros
- `movt` leaves the low bits intact.

Reading from memory

```
ldr r0, [r1] /* load r0 with data at the address in r1 */
```

- **The square brackets mean access memory at the address stored in the register.**
- In C, this would be a pointer dereference: `r0 = *r1;`

Reading memory with an offset

```
ldr r0, [r1,#4] /* load r0 from the address r1+4 */
```

- **The values r1 and #4 are added together to get the address** from which the value is loaded.
- In C, this is array indexing:

```
// assume r1 is an int* such that sizeof(r1) == 4
r0 = r1[1];
```

Writing to memory

```
str r0, [r1] /* store r0 into the address in r1 */
str r0, [r1,#4] /* store r0 into [r1+4] */
```

- Similar syntax to ldr.
- The destination is the second argument!

Loading 32-bit values: via the LDR “pseudo-instruction”

```
/* load 0x4004B00C into r1 */
ldr r1, =0x4004B00C /* notice the = sign */
```

- The = sign tells the assembler that this is a “pseudo-instruction”.
- There is no such machine instruction capable of loading a 32-bit immediate.
- **The assembler generates the code to load this value for us.**

How does LDR r0, =imm32 work?

```
ldr r1, =0x4004B00C /* notice the = sign */
```

- The 32-bit constant is placed into a "**literal pool**" at the end of the current function (immediately after the end of code).
- The pseudo-instruction compiles into:

```
ldr r1, [pc, #A]  /* pc is the program counter */
```

where `pc+A` contains the 32-bit constant in the literal pool.

- The compiler calculates the offset *#A* such that it lines up with the correct entry in the literal pool.

Addition

```
add r1, #1         /* r1 += 1 */
add r0, r1, #1  /* r0 = r1 + 1 */
add r0, r1, r2  /* r0 = r1 + r2 */
```

The immediate value can be a maximum of 12 bits long. Larger values must be loaded into a register first.

Subtraction

```
sub r1, #1        /* r1 -= 1 */
sub r0, r1, #1  /* r0 = r1 - 1 */
sub r0, r1, r2  /* r0 = r1 - r2 */
```

The immediate is 12 bits long. Larger values must be loaded into a register first.

Labelling lines of assembly code

- Can give human-readable labels to specific lines of code (so that we can 'jump' to that line of code).
- Coding convention: labels appear in the first column, and code is indented.

```
label1:
     sub r1, #1
label2:
     sub r2, #2
     b label1 /* "branch" (jump) to label1 */
```

Loop example

C code version:

```
int i = 10;
do { /* insert code here */; i--; } while (i != 0);
```

Assembly code version:

```
     mov r0, #10  /* r0 = 10 */
myloop:
     /* insert code here */
     sub r0, #1   /* r0 -= 1 */
     cmp r0, #0   /* is r0 equal to zero? */
     bne myloop   /* branch if not equal */
```

Condition flags: the cmp instruction

```
    cmp r0, #0   /* is r0 equal to zero? */
```

- The **cmp** instruction updates the “**condition flags**” that can be later used for conditional instructions.
- The condition flags are stored in a special register called the PSR (program status register).
- cmp performs a subtraction (to compare the two operands) and updates the condition flags.

Conditional branches

Instruction	Meaning
`b label`	Branch to label
`beq label`	Branch if equal
`bne label`	Branch if not equal
`blt label`	Branch if less than
`bgt label`	Branch if greater than

Refer to http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0553a/BABEHFEF.html for the complete list.

Reference card pages 1 – 2

ARM instruction set

- Refer to the ARM instruction set when programming:

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0553a/CIHJJEIH.html

Summary

- Assembly code gives low-level control over the code that is running.
- It’s mainly used for low level and/or performance-critical parts of the program in order to take advantage of specialised instructions.